



ALGOL VERSION 4 REFERENCE MANUAL

**CDC® OPERATING SYSTEMS:
NOS 1
NOS/BE 1
SCOPE 2**

REVISION RECORD	
REVISION	DESCRIPTION
A	Manual released.
(11-01-75)	
B	Updated to reflect version 4.1 and support of AAM 2 and BAM 1.5.
(03-31-78)	
Publication No.	
60496600	

REVISION LETTERS I, O, Q AND X ARE NOT USED

© 1975, 1978
 Control Data Corporation
 Printed in the United States of America

Address comments concerning
 this manual to:

CONTROL DATA CORPORATION
Publications and Graphics Division
215 MOFFETT PARK DRIVE
SUNNYVALE, CALIFORNIA 94086

or use Comment Sheet in the
 back of this manual

LIST OF EFFECTIVE PAGES

New features, as well as changes, deletions, and additions to information in this manual are indicated by bars in the margins or by a dot near the page number if the entire page is affected. A bar by the page number indicates pagination rather than content has changed.

Page	Revision
Cover	—
Title Page	—
ii	B
iii,iv	B
v,vi	B
vii	A
viii	B
ix	B
1-1 thru 1-3	A
2-1 thru 2-42	A
2-43	B
2-44	A
2-45	A
2-46	B
2-47 thru 2-52	A
2-53, 2-54	B
2-55 thru 2-58	A
2-59	B
3-1 thru 3-6	A
3-7	B
3-8 thru 3-12	A
3-13 thru 3-15	B
3-16	A
3-17	B
3-18	A
3-19	B
3-20 thru 3-29	A
3-30	B
3-31 thru 3-37	A
3-38	B
3-39, 3-40	A
3-41	B
3-42	A
3-43	B
3-44 thru 3-47	A
3-48 thru 3-54	B
3-55, 3-56	A

Page	Revision
4-1 thru 4-4	B
5-1	A
5-2	B
6-1 thru 6-4	A
6-5	B
7-1 thru 7-3	B
8-1	A
8-2	B
8-3	A
9-1 thru 9-5	A
10-1 thru 10-5	A
10-6	B
10-7	A
11-1	B
11-2, 11-3	A
11-4, 11-5	B
12-1, 12-2	A
13-1, 13-2	A
14-1 thru 14-9	A
14-10	B
14-11 thru 14-15	A
15-1, 15-2	A
15-3 thru 15-18	B
16-1	B
16-2 thru 16-12	A
16-13	B
16-14 thru 16-17	A
17-1	B
17-2 thru 17-9	A
17-10	B
17-11 thru 17-16	A
17-17	B
17-18 thru 17-24	A
17-25, 17-26	B
17-27 thru 17-30	A
A-1	A
A-2	B
B-1 thru B-12	A

Page	Revision
B-13	B
B-14 thru B-16	A
C-1	A
C-2 thru C-4	B
C-5	A
D-1 thru D-10	A
E-1 thru E-7	B
Index-1	B
Index-2 thru 4	A
Index-5, 6	B
Index-7 thru 11	A
Index-12	B
Index-13 thru 20	A
Cmt Sheet	B
Return Env.	B
Back Cover	—

PREFACE

This manual describes the ALGOL-60 language (Version 4.1) as implemented under the following operating systems:

NOS 1 for the CONTROL DATA® CYBER Models 171, 172, 173, 174, and 175; CYBER 70 Models 71, 72, 73, and 74; and 6000 Series Computer Systems

NOS/BE 1 for the CDC® CYBER 170 Series; CYBER 70 models 71, 72, 73, and 74; and 6000 Series Computer Systems

SCOPE 2 for the CONTROL DATA CYBER 170 Model 176; CYBER 70 Model 76; and 7600 Computer Systems

ALGOL 4.1 utilizes CYBER Record Manager BAM 1.5 and AAM 2 except for the AAM 2 extended index sequential file organization and related multiple index processing capability.

The ALGOL 4 compiler operates in conjunction with the COMPASS Version 3 assembly language processor. It is assumed that the reader is conversant with some ALGOL language and is familiar with one of the computer systems listed above.

ALGOL 4 conforms closely to the definition of the international algorithmic language ALGOL-60 published in The Communications of the ACM, 1963, vol. 6 no. 1, pp. 1-17; "The Revised Report on the Algorithmic Language, ALGOL-60," and also in the International Organization for Standardization's Draft ISO Recommendation No. 1538, which includes the input/output definitions.

Chapters 2 and 3 of this manual contain the entire contents of the ALGOL-60 Revised Report and the ACM proposal for input/output,[†] printed in boldface type. Material describing Control Data modifications and extensions to this standard are printed in normal type. The name ALGOL, unless printed in boldface or otherwise qualified, refers to ALGOL 4.

Other publications of interest:

<u>Publication</u>	<u>Publication Number</u>
COMPASS Version 3 Reference Manual	60492600
FORTTRAN Extended Version 4 Reference Manual	60497800
CYBER Loader Reference Manual	60429800
CYBER Record Manager Basic Access Methods Version 1.5	60482300
CYBER Record Manager Advanced Access Methods Version 2	60499300

[†]"A Proposal for Input-Output Conventions in ALGOL-60," published in The Communications of the ACM, vol. 7 no. 5, May 1964.

<u>Publication</u>	<u>Publication Number</u>
NOS/BE 1 Reference Manual	60493800
INTERCOM Version 4 Reference Manual	60494600
NOS 1 Reference Manual (Volume 1)	60435400
NOS 1 Reference Manual (Volume 2)	60445300
NOS 1 Time-Sharing User's Reference Manual	60435500
SCOPE 2 Reference Manual	60342600

This product is intended for use only as described in this document.
Control Data cannot be responsible for the proper functioning of
undescribed features or parameters.

CDC manuals can be ordered from Control Data Literature and
Distribution Services, 8001 East Bloomington Freeway, Minneapolis,
MN 55420

CONTENTS

1.	ALGOL SYSTEM DESCRIPTION	1-1	5.	Declarations	2-42
1.1	Compiler Features	1-1	5.1	Type declarations	2-42
1.2	Compiler Package	1-2	5.2	Array declarations	2-44
1.3	Compiler Structure	1-2	5.3	Switch declarations	2-46
1.4	Library Subprograms	1-3	4.5	Procedure declarations	2-46
				Examples of Procedure Declarations	2-53
				Alphabetic Index of Definitions of Concepts and Syntactic Units	2-57
2.	LANGUAGE COMPARISON WITH THE ALGOL-60 REVISED REPORT	2-1	3.	INPUT/OUTPUT	3-1
2.1	Language Conventions	2-1	3.1	Comparison with ACM Proposal for Input/Output	3-1
	Revised Report on the Algorithmic Language ALGOL-60	2-2		A Proposal for Input-Output Conventions in ALGOL-60	3-2
	Summary	2-2		A. Formats	3-2
	Contents	2-3		A.1 Number formats	3-3
	Introduction	2-4		A.2 Other formats	3-7
1.	Structure of the Language	2-7		A.3 Format strings	3-11
1.1	Formalism for syntactic description	2-7		A.4 Summary of format codes	3-12
2.	Basic Symbols, Identifiers, Numbers, and Strings.			A.5 "Standard" format	3-13
	Basic Concepts.	2-8		B. Input and Output Procedures	3-13
2.1	Letters	2-8		B.1 General characteristics	3-13
2.2	Digits, Logical values	2-9		B.2 Horizontal and vertical control	3-17
2.3	Delimiters	2-9		B.3 Layout procedures	3-19
2.4	Identifiers	2-11		B.4 List procedures	3-23
2.5	Numbers	2-12		B.5 Input and output calls	3-25
2.6	Strings	2-14		B.6 Control procedures	3-40
2.7	Quantities, kinds and scopes	2-15		B.7 Other procedures	3-42
2.8	Values and types	2-15		C. An Example	3-43
3.	Expressions	2-15		D. Machine-dependent portions	3-44
3.1	Variables	2-15	3.2	Additional Input-Output Procedures	3-45
3.2	Function designators	2-17	3.3	Control Procedures	3-49
3.3	Arithmetic expressions	2-19	3.4	Hardware Function Procedures	3-50
3.4	Boolean expressions	2-23	3.5	Miscellaneous Procedures and Functions	3-51
3.5	Designational Expressions	2-26	3.6	Input/Output Errors	3-53
4.	Statements	2-27	3.7	End of Partition	3-54
4.1	Compound statements and blocks	2-27	3.8	Efficient Use of Formatted Input/Output	3-54
4.2	Assignment statements	2-29	3.9	Extended Core Storage, Large Core Memory Procedures	3-56
4.3	Go to statements	2-31	4.	INPUT TO COMPILATION	4-1
4.4	Dummy statements	2-32	4.1	Source Program Definition	4-1
4.5	Conditional statements	2-32	4.2	Source Procedure Definition	4-2
4.6	For statements	2-35			
4.7	Procedure statements	2-38			

4.3	Source Input Restrictions	4-3	12.	OPTIMIZATION	12-1
4.4	Language Conventions	4-3	12.1	Machine Independent Optimization	12-1
4.5	Source Line Conventions	4-4	12.2	Vector Functions	12-2
4.6	Source Deck	4-4	12.3	Efficient Programming for Run-Time Performance	12-2
5.	OUTPUT FROM COMPILATION	5-1	13.	EXECUTION TIME ABNORMAL TERMINATION DUMP	13-1
5.1	Binary Output	5-1	13.1	Structured Dump	13-1
5.2	Assembly Language Object Code	5-1	13.2	Environmental Information	13-1
5.3	Source Listing	5-1	13.3	Cross-Reference Listing	13-2
6.	ALGOL CONTROL STATEMENT	6-1	14.	RUN-TIME DESCRIPTION	14-1
6.1	Control Card Parameters	6-1	14.1	Object Program and Stacks	14-1
6.2	Restrictions and Errors	6-5	14.2	Stack Entries	14-5
6.3	Compilation and Execution Field Length	6-5	14.3	Details of Descriptors	14-8
7.	CHANNEL STATEMENTS	7-1	15.	DIAGNOSTICS	15-1
7.1	Channel Define Statement	7-1	15.1	Compiler Diagnostics	15-1
7.2	Channel Equate Statement	7-2	15.2	Object Time Diagnostics	15-4
7.3	Duplication of Channel Numbers	7-2	15.3	ALGEDIT Diagnostics	15-4
7.4	Duplication of File Names	7-3	16.	ALGOL INTERACTIVE DEBUGGING AIDS	16-1
7.5	Standard ALGOL Channel Statements	7-3	16.1	Interactive Mode	16-1
7.6	Typical Channel Statements	7-3	16.2	Interactive Commands	16-3
8.	RUN TIME OPTIONS	8-1	16.3	Fatal Errors	16-13
8.1	Options	8-1	16.4	Using AIDA in Batch Mode	16-13
8.2	Example	8-3	16.5	Example	16-14
9.	DEBUGGING FACILITIES	9-1	17.	ALGEDIT	17-1
9.1	General Description	9-1	17.1	ALGOL Interactive Syntax Checker	17-1
9.2	Debugging Directives	9-1	17.2	ALGEDIT Command Syntax	17-2
9.3	Monitoring Messages	9-3	17.3	Special Characters	17-2
9.4	Monitoring of Simple Variables	9-3	17.4	Accessing ALGEDIT	17-2
9.5	Monitoring of Arrays	9-4	17.5	Exiting from ALGEDIT	17-5
9.6	Monitoring of Labels	9-5	17.6	File Creation and Modification	17-6
9.7	Monitoring of Procedures	9-5	17.7	File Storage and Display	17-17
10.	OVERLAYS	10-1	17.8	Syntax Checking	17-23
10.1	Overlay Declarations	10-1	17.9	Compilation and Execution	17-26
10.2	Example	10-1	17.10	Example	17-28
10.3	Semantics	10-1			
10.4	Restrictions	10-2			
10.5	Examples	10-3			
11.	USE OF LCM AND ECS	11-1			
11.1	Virtual Arrays and S Option	11-1			
11.2	Data Stacks in ECS/LCM	11-2			

APPENDIXES

A.	STANDARD CHARACTER SETS	A-1	D.	STANDARD PROCEDURES FOR VECTOR AND MATRIX MANIPULATIONS	D-1
B.	INTERFACE MACROS	B-1			
C.	HARDWARE REPRESENTATION OF ALGOL SYMBOLS	C-1	E.	GLOSSARY	E-1

1.1 COMPILER FEATURES

The ALGOL 4 compiler for the CYBER series computers is based in design on the ALGOL compiler developed by Regnecentralen, Copenhagen, Denmark, for the GIER computer. This design has been adopted and, to a great degree, modified and extended by Control Data to provide the most generally advantageous features for an ALGOL compiler.

These features include:

- Implementation of the complete ALGOL-60 revised language (wherever feasible and not in conflict with other advantages)

- Comprehensive input/output procedures

- Extensive compile time and object time diagnostics

- Wide variety of compilation options, such as the ability to compile both programs and procedures

- Ability to generate and execute the object program in either overlay or nonoverlay form

- Optimization facilities

- Optional use of Large Core Memory and Extended Core Storage for the storage of arrays

SOURCE INPUT

The source input for compilation can be specified from any device by an ALGOL control statement option. Source input can consist of both source programs and source procedures. More than one source program or source procedure may be compiled with a single call.

COMPILE-TIME ERROR DETECTION

The compiler detects all source language errors and lists a diagnostic for each. The compiler also incorporates further checking into the object program to detect program errors that can be found only at execution time. Compilation always proceeds to the end of the source deck with normal error checking regardless of the occurrence of a source language error, but object code generation is suppressed if any errors are detected during compilation.

COMPILER OUTPUTS

Compiler output is normally listed on the standard system file OUTPUT. Output also may be requested on a different file with an ALGOL control statement option. The object code is in standard relocatable binary format.

OPTIMIZATION FACILITIES

The user has the option of requesting optimization of a program at both the source language and machine code levels.

OBJECT PROGRAM EXECUTION

Execution of the object program is controlled by the run time system, which is external to the generated program.

OBJECT ERROR DETECTION

The object program includes code which detects errors not detected during compilation. If an error occurs, an error message is issued, a symbolic dump if selected is printed, and the run is terminated. The dump displays current values of declared variables in a form easily related to the source program.

1.2 COMPILER PACKAGE

The ALGOL compiler package consists of the following subprograms recorded on the system library:

The compiler: ALGOL, ALG0, ALG1, ALG2, ALG3, ALG4, ALG5, and ALG6.

The library subprograms which are available to object programs generated by the compiler.

1.3 COMPILER STRUCTURE

ALGOL is the internal controller of the compiler and its main function is to load and pass control to each subprogram as required.

ALG0 processes the control statement options delivered by the operating system.

ALG1 through ALG6 each form one overlay of the compiler. Each subprogram overlays the previous one in a separate load. Each overlay generates an intermediate form of the source text which is used as input to the next overlay.

ALG1 performs syntactic analysis of source text. If any fatal errors have been encountered, compilation terminates after this pass, providing the F option has been selected.

ALG2 performs semantic analysis of the source text. If any fatal errors have been encountered in this pass or in ALG1, compilation terminates after this overlay.

ALG3 performs checking of actual and formal procedure parameters and starts source language optimization. For any nonfatal errors, a warning message is printed. No fatal errors are detected by this pass. If no optimization has been requested, this overlay is not called and compilation proceeds directly from ALG2 to ALG4.

ALG4 completes source language optimization and performs the source text translation in a form suitable for code generation in ALG5.

ALG5 produces final output from the compiler, including the object code in standard relocatable binary format.

ALG6 produces the cross-reference map and creates the dumpfile to be used at execution time by the dump routine. This overlay is called only if the R or D option is selected.

1.4 LIBRARY SUBPROGRAMS

The library subprograms contain all standard procedures, which can be called without prior declaration in an ALGOL source text. They also contain subprograms to perform object-time control functions external to the generated object program.

LANGUAGE COMPARISON WITH THE ALGOL-60 REVISED REPORT

2

2.1 LANGUAGE CONVENTIONS

In this manual, ALGOL is described in terms of two languages: reference and hardware. These representations are discussed in the introduction to the ALGOL-60 Revised Report.

The reference language is independent of computers, operating systems, or compiler implementation, and uses a set of basic symbols to define the syntax and semantics of ALGOL. In particular, a reference language convention uses underlining, such as in begin, to denote independent basic symbols that have no relation to the individual letters of which they are composed.

The hardware language is a representation of ALGOL symbols that the computer can recognize; this is the language used by the programmer. For example, where the reference language calls for the basic ALGOL symbol begin, the programmer punches the seven hardware characters ≠BEGIN≠ or "BEGIN". The hardware representations of ALGOL symbols are shown in Appendix C.

Both the reference language and the hardware language are used in this manual. Only the reference language is used in chapters 2 and 3; both are used elsewhere in the manual.

All descriptions of language modifications occur at the main reference in the Report; when feasible, language modifications are also noted at other points of reference. The reader should assume that modifications apply to all references to the features, noted or otherwise. If no comments appear at the main reference in the Report regarding language modifications to a particular section or feature, it is implemented in full accordance with the Report.

Not included in the language description in this chapter are reserved identifiers referencing input/output procedures, described in Chapter 3.

The ALGOL-60 Revised Report as published in The Communications of the ACM, vol. 6, no. 1, pp 1-17 follows. Wherever Control Data's implementation of the language differs from the Report, the Report is printed first in boldface and the Control Data modification follows in standard type.

REVISED REPORT ON THE ALGORITHMIC LANGUAGE ALGOL-60[†]

Peter Naur (Editor)

J. W. Backus
F. L. Bauer
J. Green

C. Katz
J. McCarthy
A. J. Perlis

H. Rutishauser
K. Samelson
B. Vauquois

J. H. Wegstein
A. van Wijngaarden
M. Woodger

Dedicated to the memory of William Turanski.

SUMMARY

The report gives a complete defining description of the international algorithmic language ALGOL-60. This is a language suitable for expressing a large class of numerical processes in a form sufficiently concise for direct automatic translation into the language of programmed automatic computers.

The introduction contains an account of the preparatory work leading up to the final conference, where the language was defined. In addition, the notions, reference language, publication language and hardware representations are explained.

In the first chapter, a survey of the basic constituents and features of the language is given, and the formal notation, by which the syntactic structure is defined, is explained.

The second chapter lists all the basic symbols, and the syntactic units known as identifiers, numbers and strings are defined. Further, some important notions such as quantity and value are defined.

The third chapter explains the rules for forming expressions and the meaning of these expressions. Three different types of expressions exist: arithmetic, Boolean (logical) and designational.

The fourth chapter describes the operational units of the language, known as statements. The basic statements are: assignment statements (evaluation of a formula), go to statements (explicit break of the sequence of execution of statements), dummy statements, and procedure statements (call for execution of a closed process, defined by a procedure declaration). The formation of more complex structures, having statement character, is explained. These include: conditional statements, for statements, compound statements, and blocks.

In the fifth chapter, the units known as declarations, serving for defining permanent properties of the units entering into a process described in the language, are defined.

The report ends with two detailed examples of the use of the language and an alphabetic index of definitions.

[†]This report is published in The Communications of the ACM, in Numerische Mathematik, and in the Computer Journal.

CONTENTS

Introduction

- 1. Structure of the Language**
 - 1.1 Formalism for syntatic description**
- 2. Basic Symbols, Identifiers, Numbers, and Strings. Basic Concepts**
 - 2.1 Letters**
 - 2.2 Digits, Logical values**
 - 2.3 Delimiters**
 - 2.4 Identifiers**
 - 2.5 Numbers**
 - 2.6 Strings**
 - 2.7 Quantities, kinds and scopes**
 - 2.8 Values and types**
- 3. Expressions**
 - 3.1 Variables**
 - 3.2 Function designators**
 - 3.3 Arithmetic expressions**
 - 3.4 Boolean expressions**
 - 3.5 Designational expressions**
- 4. Statements**
 - 4.1 Compound statements and blocks**
 - 4.2 Assignment statements**
 - 4.3 Go to statements**
 - 4.4 Dummy statements**
 - 4.5 Conditional statements**
 - 4.6 For statements**
 - 4.7 Procedure statements**
- 5. Declarations**
 - 5.1 Type declarations**
 - 5.2 Array declarations**
 - 5.3 Switch declarations**
 - 5.4 Procedure declarations**

Examples of Procedure Declarations

Alphabetic Index of Definitions of Concepts and Syntatic Units

INTRODUCTION

Background

After the publication of a preliminary report on the algorithmic language ALGOL[†], as prepared at a conference in Zurich in 1958, much interest in the ALGOL language developed.

As a result of an informal meeting held at Mainz in November 1958, about forty interested persons from several European countries held an ALGOL implementation conference in Copenhagen in February 1959. A "hardware group" was formed for working cooperatively right down to the level of the paper tape code. This conference also led to the publication by Regnecentralen, Copenhagen, of an ALGOL Bulletin, edited by Peter Naur, which served as a forum for further discussion. During the June 1959 ICIP Conference in Paris several meetings, both formal and informal ones, were held. These meetings revealed some misunderstandings as to the intent of the group which was primarily responsible for the formulation of the language, but at the same time made it clear that there exists a wide appreciation of the effort involved. As a result of the discussions it was decided to hold an international meeting in January 1960 for improving the ALGOL language and preparing a final report. At a European ALGOL Conference in Paris in November 1959 which was attended by about fifty people, seven European representatives were selected to attend the January 1960 Conference, and they represented the following organizations: Association Francaise de Calcul, British Computer Society, Gesellschaft für Angewandte Mathematik und Mechanik, and Nederlands Rekenmachine Genootschap. The seven representatives held a final preparatory meeting at Mainz in December 1959.

Meanwhile, in the United States, anyone who wished to suggest changes or corrections to ALGOL was requested to send his comments to the Communications of the ACM, where they were published. These comments then became the basis of consideration for changes in the ALGOL language. Both the SHARE and USE organizations established ALGOL working groups, and both organizations were represented on the ACM Committee on Programming Languages. The ACM Committee met in Washington in November 1959 and considered all comments on ALGOL that had been sent to the ACM Communications. Also, seven representatives were selected to attend the January 1960 international conference. These seven representatives held a final preparatory meeting in Boston in December 1959.

January 1960 Conference

The thirteen representatives, from Denmark, England, France, Germany, Holland, Switzerland, and the United States, conferred in Paris from January 11 to 16, 1960. Prior to this meeting a completely new draft report was worked out from the preliminary report and the recommendations of the preparatory meetings by Peter Naur and the conference adopted this new form as the basis for its report. The Conference then proceeded to work for agreement on each item of the report. The present report represents the union of the Committee's concepts and the intersection of its agreement.

April 1962 Conference (Edited by M. Woodger)

A meeting of some of the authors of ALGOL-60 was held on April 2-3, 1962 in Rome, Italy, through the facilities and courtesy of the International Computation Centre.

[†]Preliminary report — International Algebraic Language. Comm ACM1, 12 (1958), 8.

Report on the Algorithmic Language ALGOL by the ACM Committee on Programming Languages, edited by A. J. Perlis and K. Samelson. Num, Math. 1 (1959), 41-60.

The following were present:

Authors	Advisers	Observer
F. L. Bauer	M. Paul	W. L. van der Poel (Chairman IFIP TC 2.1 Working Group ALGOL)
J. Green	R. Franciotti	
C. Katz	P. Z. Ingerman	
R. Kogon		
(representing J. W. Backus)		
P. Naur		
K. Samelson	G. Seegmüller	
J. H. Wegstein	R. E. Utman	
A. van Wijngaarden		
M. Woodger	P. Landin	

The purpose of the meeting was to correct known errors in, attempt to eliminate apparent ambiguities in, and otherwise clarify the ALGOL-60 Report. Extensions to the language were not considered at the meeting. Various proposals for correction and clarification that were submitted by interested parties in response to the Questionnaire in ALGOL Bulletin No. 14 were used as a guide.

This report constitutes a supplement to the ALGOL-60 Report which should resolve a number of difficulties therein. Not all of the questions raised concerning the original report could be resolved. Rather than risk hastily drawn conclusions on a number of subtle points, which might create new ambiguities, the committee decided to report only those points which they unanimously felt could be stated in clear and unambiguous fashion.

Questions concerned with the following areas are left for further consideration by Working Group 2.1 of IFIP, in the expectation that current work on advanced programming languages will lead to better resolution:

1. Side effects of functions
2. The call by name concept
3. own: static or dynamic
4. For statement: static or dynamic
5. Conflict between specification and declaration

The authors of the ALGOL Report present at the Rome Conference, being aware of the formation of a Working Group on ALGOL by IFIP, accepted that any collective responsibility which they might have with respect to the development, specification and refinement of the ALGOL language will from now on be transferred to that body.

This report has been reviewed by IFIP TC 2 on Programming Languages in August 1962 and has been approved by the Council of the International Federation for Information Processing.

As with the preliminary ALGOL report, three different levels of language are recognized, namely a Reference Language, a Publication Language and several Hardware Representations.

REFERENCE LANGUAGE

1. It is the working language of the committee.
2. It is the defining language.
3. The characters are determined by ease of mutual understanding and not by any computer limitations, coders notation, or pure mathematical notation.
4. It is the basic reference and guide for compiler builders.
5. It is the guide for all hardware representations.
6. It is the guide for transliterating from publication language to any locally appropriate hardware representations.
7. The main publications of the ALGOL language itself will use the reference representation.

PUBLICATION LANGUAGE

1. The publication language admits variations of the reference language according to usage of printing and handwriting (e.g., subscripts, spaces, exponents, Greek letters).
2. It is used for stating and communicating processes.
3. The characters to be used may be different in different countries but univocal correspondence with reference representation must be secured.

HARDWARE REPRESENTATIONS

1. Each one of these is a condensation of the reference language enforced by the limited number of characters on standard input equipment.
2. Each one of these uses the character set of a particular computer and is the language accepted by a translator for that computer.
3. Each one of these must be accompanied by a special set of rules for transliterating from Publication or Reference language.

For transliteration between the reference language, and a language suitable for publication, among others, the following rules are recommended.

Reference Language

Publication Language

Subscript bracket []	Lowering of the line between the brackets and removal of the brackets
Exponentiation ↑	Raising of the exponent
Parentheses ()	Any-form of parentheses, brackets, braces
Basis of ten 10	Raising of the ten and of the following integral number, inserting of the intended multiplication sign.

DESCRIPTION OF THE REFERENCE LANGUAGE

1. Structure of the Language

As stated in the introduction, the algorithmic language has three different kinds of representations—reference, hardware, and publication—and the development described in the sequel is in terms of the reference representation. This means that all objects defined within the language are represented by a given set of symbols—and it is only in the choice of symbols that the other two representations may differ. Structure and content must be the same for all representations.

The purpose of the algorithmic language is to describe computational processes. The basic concept used for the description of calculating rules is the well-known arithmetic expression containing as constituents numbers, variables, and functions. From such expressions are compounded, by applying rules of arithmetic composition, self-contained units of the language—explicit formulae—called assignment statements.

To show the flow of computational processes, certain nonarithmetic statements and statement clauses are added which may describe, e.g., alternatives, or iterative repetitions of computing statements. Since it is necessary for the function of these statements that one statement refer to another, statements may be provided with labels. A sequence of statements may be enclosed between the statement brackets begin and end to form a compound statement.

Statements are supported by declarations which are not themselves computing instructions but inform the translator of the existence and certain properties of objects appearing in statements, such as the class of numbers taken on as values by a variable, the dimension of an array of numbers, or even the set of rules defining a function. A sequence of declarations followed by a sequence of statements and enclosed between begin and end constitutes a block. Every declaration appears in a block in this way and is valid only for that block.

A program is a block or compound statement which is not contained within another statement and which makes no use of other statements not contained within it.

In the sequel the syntax and semantics of the language will be given.[†]

1.1 FORMALISM FOR SYNTACTIC DESCRIPTION

The syntax will be described with the aid of metalinguistic formulae.[‡] Their interpretation is best explained by an example

$$\langle ab \rangle ::= ([| \langle ab \rangle (| \langle ab \rangle \langle d \rangle$$

[†]Whenever the precision of arithmetic is stated as being in general not specified, or the outcome of a certain process is left undefined, or said to be undefined, this is to be interpreted in the sense that a program only fully defines a computational process if the accompanying information specifies the precision assumed, the kind of arithmetic assumed, and the course of action to be taken in all such cases as may occur during the execution of the computation.

[‡]Cf. J.W. Backus, The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM conference. Proc. Internat. Conf. Inf. Proc., UNESCO, Paris, June 1959.

Sequences of characters enclosed in the brackets $\langle \rangle$ represent metalinguistic variables whose values are sequences of symbols. The mark $::=$ and $|$ (the latter with the meaning of or) are metalinguistic connectives. Any mark in a formula, which is not a variable or a connective, denotes itself (or the class of marks which are similar to it). Juxtaposition of marks and/or variables in a formula signifies juxtaposition of the sequences denoted. Thus the formula above gives a recursive rule for the formation of values of the variable $\langle ab \rangle$. It indicates that $\langle ab \rangle$ may have the value (or $|$ or that given some legitimate value of $\langle ab \rangle$, another may be formed by following it with the character (or by following it with some value of the variable $\langle d \rangle$. If the values of $\langle d \rangle$ are the decimal digits, some values of $\langle ab \rangle$ are:

```

[(((1(37(
(12345(
((
[86

```

In order to facilitate the study, the symbols used for distinguishing the metalinguistic variables (i.e., the sequences of characters appearing within the brackets $\langle \rangle$ as ab in the above example) have been chosen to be words describing approximately the nature of the corresponding variable. Where words which have appeared in this manner are used elsewhere in the text they will refer to the corresponding syntactic definition. In addition some formulae have been given in more than one place.

Definition:

$\langle \text{empty} \rangle ::=$

(i.e., the null string of symbols).

2. Basic Symbols, Identifiers, Numbers, and Strings. Basic Concepts.

The reference language is built up from the following basic symbols:

$\langle \text{basic symbol} \rangle ::= \langle \text{letter} \rangle | \langle \text{digit} \rangle | \langle \text{logical value} \rangle | \langle \text{delimiter} \rangle$

2. Basic Symbols, Identifiers, Numbers, and Strings. Basic Concepts.

Other available characters not used in the hardware representation are defined to be extra basic symbols. They may occur only within strings and comment sequences.

2.1 LETTERS

$\langle \text{letter} \rangle ::= a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z |$

$A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z$

This alphabet may arbitrarily be restricted, or extended with any other distinctive character (i.e., character not coinciding with any digit, logical value or delimiter). Letters do not have individual meaning. They are used for forming identifiers and strings[†] (Cf. sections 2.4 Identifiers, 2.6 Strings).

2.1 Letters

Since there is hardware representation for upper case letters only, lower case letters have no meaning.

2.2.1 DIGITS

$\langle \text{digit} \rangle ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

Digits are used for forming numbers, identifiers, and strings.

2.2.2 LOGICAL VALUES

$\langle \text{logical value} \rangle ::= \underline{\text{true}} \mid \underline{\text{false}}$

The logical values have a fixed obvious meaning.

2.3 DELIMITERS

$\langle \text{delimiter} \rangle ::= \langle \text{operator} \rangle \mid \langle \text{separator} \rangle \mid \langle \text{bracket} \rangle \mid \langle \text{declarator} \rangle \mid \langle \text{specifier} \rangle$

$\langle \text{operator} \rangle ::= \langle \text{arithmetic operator} \rangle \mid \langle \text{relational operator} \rangle \mid \langle \text{logical operator} \rangle \mid$

$\langle \text{sequential operator} \rangle$

$\langle \text{arithmetic operator} \rangle ::= + \mid - \mid \times \mid / \mid \div \mid \uparrow$

$\langle \text{relational operator} \rangle ::= < \mid \leq \mid = \mid \geq \mid > \mid \neq$

$\langle \text{logical operator} \rangle ::= \equiv \mid \supset \mid \vee \mid \wedge \mid \neg$

$\langle \text{sequential operator} \rangle ::= \underline{\text{go to}} \mid \underline{\text{if}} \mid \underline{\text{then}} \mid \underline{\text{else}} \mid \underline{\text{for}} \mid \underline{\text{do}}^{\ddagger}$

$\langle \text{separator} \rangle ::= , \mid . \mid 10 \mid : \mid ; \mid := \mid _ \mid \underline{\text{step}} \mid \underline{\text{until}} \mid \underline{\text{while}} \mid \underline{\text{comment}}$

[†]It should be particularly noted that throughout the reference language underlining is used for defining independent basic symbols (see sections 2.2.2 and 2.3). These are understood to have no relation to the individual letters of which they are composed. Within the present report (not including headings) underlining will be used for no other purpose.

[‡]do is used in for statements. It has no relation whatsoever to the do of the preliminary report, which is not included in ALGOL-60.

< bracket > ::= (|) | [|] | ' | ' | begin | end

< declarator > ::= own | Boolean | integer | real | array | switch | procedure

< specifier > ::= string | label | value

Delimiters have a fixed meaning which for the most part is obvious or else will be given at the appropriate place in the sequel.

Typographical features such as blank space or change to a new line have no significance in the reference language. They may, however, be used freely for facilitating reading. For the purpose of including text among the symbols of a program the following "comment" conventions hold:

The sequence of basic symbols:	is equivalent to:
<u>;</u> <u>comment</u> < any sequence not containing <u>;</u> >	<u>;</u>
<u>begin</u> <u>comment</u> < any sequence not containing <u>;</u> >	<u>begin</u>
<u>end</u> < any sequence not containing <u>end</u> or <u>;</u> or <u>else</u> >	<u>end</u>

By equivalence is here meant that any of the three structures shown in the left hand column may be replaced, in any occurrence outside of strings, by the symbol shown on the same line in the right hand column without any effect on the action of the program. It is further understood that the comment structure encountered first in the text when reading from left to right has precedence in being replaced over later structures contained in the sequence.

2.3 Delimiters

The symbols code, algol and fortran are added to the language to permit reference to separately compiled procedures (section 5.4.6).

< code procedure body indicator > ::= code | algol | fortran

2.3.1 Additional Delimiters

Additional delimiters are not added to the language, but appear in comments. Under the normal mode of compilation, these delimiters are ignored by the compiler and thus have no effect during the execution of the resulting object code. Since these delimiters and the directives of which they form a part are acceptable ALGOL-60 character sequences, it is unnecessary to remove them from a source program submitted to an ALGOL compiler in which they are not recognizable. However, if the C option is selected (chapter 6), any additional delimiters are detected by the compiler and cause the corresponding actions to be taken for the object program. The virtual array directive, however, is always detected by the compiler and is not dependent on the C option.

The additional delimiters are:

- for the debugging facility – trace, snap, snapoff (chapter 9)
- for the overlay directive – overlay (chapter 10)
- for the virtual array directive – virtual (chapter 6 – S option, and chapter 11)
- for the array bound checking directives – checkon and checkoff (chapter 6)

2.4 IDENTIFIERS

2.4.1 SYNTAX

< identifier > ::= < letter > | < identifier > < letter > | < identifier > < digit >

2.4.1 Syntax

External identifier is added to the language and is defined as follows:

<external identifier> ::= <identifier with length less than or equal to 7 letters and digits beginning with a letter>

2.4.2 EXAMPLES

**q
Soup
V 17a
a34kTMNs
MARILYN**

2.4.3 SEMANTICS

Identifiers have no inherent meaning, but serve to the identification of simple variables, arrays, labels, switches, and procedures. They may be chosen freely (cf., however, section 3.2.4 Standard Functions).

The same identifier cannot be used to denote two different quantities except when these quantities have disjoint scopes as defined by the declarations of the program (cf. section 2.7. Quantities, Kinds and Scopes, and section 5., Declarations).

2.4.3 Semantics

The maximum number of characters of an identifier used for unique identification is 63. Identifiers that differ in the first 63 characters are distinguished while those that do not differ are not distinguished.

The maximum number of identifiers that can be handled by the compiler without causing identifier table overflow depends on the identifier sizes and the amount of memory available to the compiler.

The compiler itself can handle up to 4000 unique identifiers. Identifier table overflow generally occurs before this number is reached.

2.5 NUMBERS

2.5.1 SYNTAX

$\langle \text{unsigned integer} \rangle ::= \langle \text{digit} \rangle | \langle \text{unsigned integer} \rangle \langle \text{digit} \rangle$

$\langle \text{integer} \rangle ::= \langle \text{unsigned integer} \rangle | + \langle \text{unsigned integer} \rangle | - \langle \text{unsigned integer} \rangle$

$\langle \text{decimal fraction} \rangle ::= . \langle \text{unsigned integer} \rangle$

$\langle \text{exponent part} \rangle ::= 10 \langle \text{integer} \rangle$

$\langle \text{decimal number} \rangle ::= \langle \text{unsigned integer} \rangle | \langle \text{decimal fraction} \rangle |$

$\langle \text{unsigned integer} \rangle \langle \text{decimal fraction} \rangle$

$\langle \text{unsigned number} \rangle ::= \langle \text{decimal number} \rangle | \langle \text{exponent part} \rangle |$

$\langle \text{decimal number} \rangle \langle \text{exponent part} \rangle$

$\langle \text{number} \rangle ::= \langle \text{unsigned number} \rangle | + \langle \text{unsigned number} \rangle |$

$- \langle \text{unsigned number} \rangle$

2.5.2 EXAMPLES

0	-200.084	-.083 10 -02
177	+07.43 10 8	-10 7
.5384	9.34 10 +10	10 -4
+0.7300	2 10 -4	+10 +5

2.5.3 SEMANTICS

Decimal numbers have their conventional meaning. The exponent part is a scale factor expressed as an integral power of 10.

2.5.3 Semantics

A number has the format

$$d_1 d_2 \dots d_j . d_{j+1} d_{j+2} \dots d_n 10^{\pm e_1 e_2 \dots e_m}$$

where the decimal point and the exponent field are not necessarily explicit. If the decimal point is not explicit, it is assumed to follow the digit d_n ($j=n$). If the exponent field is not explicit, zero value is assumed. If the sign of the exponent field is not explicit, a positive exponent is assumed. Thus, all numbers are considered to have the same format and are treated identically.

A number is modified in three steps before it is converted to its final internal representation. (section 5.1.3)

1. Leading zeros are eliminated, including any following the decimal point.
2. The digits following the fifteenth non-zero digit are discarded. The number of digits discarded is added to the value of the exponent field.
3. The effect of the decimal point is incorporated by subtracting $n-j$ (number of digits to the right of the point) from the value of the exponent field.

These three modifications effectively produce a number of the form $d_i d_{i+1} \dots d_k 10^e$ where d_i is the first non-zero digit in the original number. If no non-zero digit is found, the number is given the internal value 0.

The last significant digit, d_k , is d_n if $n < i+14$ or is d_{i+14} otherwise.

The resulting exponent field value, e , is given by:

$$e = e_1 e_2 \dots e_m - (n-j) + \max(0, n-(i+14)).$$

If a real or integer number is larger than the largest number acceptable to the compiler, it is replaced by the largest number (section 5.1.3). In the case of integer numbers, the largest number is stored as real by appending .0 at the end and converting to floating-point representation.

If the absolute value of a real number is smaller than the absolute value of the smallest real number distinguishable from zero (section 5.1.3), the compiler replaces it by the smallest number.

In all cases of overflow or underflow, a warning message is given.

The values of the largest number acceptable and the smallest real number distinguishable from zero are available via the standard procedure THRESHOLD (chapter 3, section 3.5).

Real numbers are evaluated to full accuracy before rounding.

While integers and numbers are defined in this section, only unsigned integers and unsigned numbers are used elsewhere in the reference document and are recognized by the compiler. Signed numbers are treated as expressions.

2.5.4 TYPES

Integers are of type integer. All other numbers are of type real (cf. Section 5.1 Type Declarations).

2.5.4 Types

During compilation, numbers are flagged as type real or integer, according to the following rules:

Any number with an explicit decimal point or an explicit exponent part is flagged as real.

All other numbers are flagged as integer.

Real and integer numbers are represented internally in the same form as real and integer variables (section 5.1.3).

Signed numbers are treated as expressions.

During object code generation, the type and value of a number is sometimes changed depending on the operation with which it is associated in the source program (section 3.3.4 and 4.3.4).

2.6 STRINGS

2.6.1 SYNTAX

< proper string > ::= < any sequence of basic symbols not containing ' or ' > | < empty >

< open string > ::= < proper string > | '< open string >' |

< open string > < open string >

< string > ::= '< open string >'

2.6.1 Syntax

A proper string is defined as follows:

< proper string > ::= < empty > | < any sequence of characters not containing the symbols ' or ' >

The syntax of < open string > is replaced by:

< open string > ::= < proper string > | < open string > < string > < open string >

2.6.2 EXAMPLES

```
'5k,-' [ [ [ '^ = / : 'Tt'
'.This□ is□ a □ 'string''
```

2.6.3 SEMANTICS

In order to enable the language to handle arbitrary sequences of basic symbols the string quotes ' and ' are introduced. The symbol □ denotes a space. It has no significance outside strings.

Strings are used as actual parameters of procedures (cf. sections 3.2. Function Designators and 4.7. Procedure Statements).

2.6.3 Semantics

The string quote symbols ' and ' are introduced to enable the language to handle arbitrary sequences of allowable characters (not basic symbols, as defined in the Report). See appendix C for the hardware representation of the string quote symbols.

2.7 QUANTITIES, KINDS AND SCOPES

The following kinds of quantities are distinguished: simple variables, arrays, labels, switches, and procedures.

The scope of a quantity is the set of statements and expressions in which the declaration of the identifier associated with that quantity is valid. For labels, see section 4.1.3.

2.8 VALUES AND TYPES

A value is an ordered set of numbers (special case: a single number), an ordered set of logical values (special case: a single logical value), or a label.

Certain of the syntactic units are said to possess values. These values will in general change during the execution of the program. The values of expressions and their constituents are defined in section 3. The value of an array identifier is the ordered set of values of the corresponding array of subscripted variables (cf. section 3.1.4.1).

The various "types" (integer, real, Boolean) basically denote properties of values. The types associated with syntactic units refer to the values of these units.

3. Expressions

In the language the primary constituents of the programs describing algorithmic processes are arithmetic, Boolean, and designational expressions. Constituents of these expressions, except for certain delimiters, are logical values, numbers, variables, function designators, and elementary arithmetic, relational, logical, and sequential operators. Since the syntactic definition of both variables and function designators contains expressions, the definition of expressions, and their constituents, is necessarily recursive.

$\langle \text{expression} \rangle ::= \langle \text{arithmetic expression} \rangle \mid \langle \text{Boolean expression} \rangle \mid \langle \text{designational expression} \rangle$

3. Expressions

Additional constituents of expressions are labels and switch designators.

3.1 VARIABLES

3.1.1 SYNTAX

$\langle \text{variable identifier} \rangle ::= \langle \text{identifier} \rangle$

$\langle \text{simple variable} \rangle ::= \langle \text{variable identifier} \rangle$

$\langle \text{subscript expression} \rangle ::= \langle \text{arithmetic expression} \rangle$

$\langle \text{subscript list} \rangle ::= \langle \text{subscript expression} \rangle \mid \langle \text{subscript list} \rangle , \langle \text{subscript expression} \rangle$

< array identifier > ::= < identifier >

< subscripted variable > ::= < array identifier > [< subscript list >]

< variable > ::= < simple variable > | < subscripted variable >

3.1.2 EXAMPLES

```
epsilon
detA
a17
Q [7,2]
x[sin(n×pi/2), Q [3, n, 4] ]
```

3.1.3 SEMANTICS

A variable is a designation given to a single value. This value may be used in expressions for forming other values and may be changed at will by means of assignment statements (section 4.2). The type of the value of a particular variable is defined in the declaration for the variable itself (cf. section 5.1. Type Declarations) or for the corresponding array identifier (cf. section 5.2 Array Declarations).

3.1.4 SUBSCRIPTS

3.1.4.1 Subscripted variables designate values which are components of multidimensional arrays (cf. section 5.2 Array Declarations). Each arithmetic expression of the subscript list occupies one subscript position of the subscripted variable, and is called a subscript. The complete list of subscripts is enclosed in the subscript brackets []. The array component referred to by a subscripted variable is specified by the actual numerical value of its subscripts (cf. section 3.3 Arithmetic Expressions).

3.1.4.2 Each subscript position acts like a variable of type integer and the evaluation of the subscript is understood to be equivalent to an assignment to this fictitious variable (cf. section 4.2.4). The value of the subscripted variable is defined only if the value of the subscript expression is within the subscript bounds of the array (cf. section 5.2 Array Declarations).

3.1.4.2 The subscript expressions are evaluated in sequence from left to right.

An optional check is available for each subscript expression (see C option, chapter 6, section 6.1.2). If an attempt is made to access a subscripted variable with a subscript expression outside the corresponding subscript bounds of the array, an error is signalled if the check is operational but the effect is undefined if the check is inhibited.

3.2 FUNCTION DESIGNATORS

3.2.1 SYNTAX

`< procedure identifier > ::= < identifier >`

`< actual parameter > ::= < string > | < expression > | < array identifier > |`
`< switch identifier > | < procedure identifier >`

`< letter string > ::= < letter > | < letter string > < letter >`

`< parameter delimiter > ::= , |) < letter string > : (`

`< actual parameter list > ::= < actual parameter > |`
`< actual parameter list > < parameter delimiter > < actual parameter >`

`< actual parameter part > ::= < empty > | (< actual parameter list >)`

`< function designator > ::= < procedure identifier > < actual parameter part >`

3.2.2 EXAMPLES

<code>sin (a-b)</code>	<code>S(s-5) Temperature: (T) Pressure: (P)</code>
<code>J (v+s,n)</code>	<code>Compile(':=') Stack: (Q)</code>
<code>R</code>	

3.2.3 SEMANTICS

Function designators define single numerical or logical values, which result through the application of given sets of rules defined by a procedure declaration (cf. section 5.4. Procedure Declarations) to fixed sets of actual parameters. The rules governing specification of actual parameters are given in section 4.7. Procedure Statements. Not every procedure declaration defines the value of a function designator.

3.2.3 Semantics

There is no restriction on side-effects, e.g., on the alteration of the values of non-local variables during the evaluation of a function designator.

It is always permissible to leave the body of a procedure which defines a function designator via a go to statement. In such a case, execution of the statement containing the function designator is discontinued.

3.2.4 STANDARD FUNCTIONS

Certain identifiers should be reserved for the standard functions of analysis which will be expressed as procedures. It is recommended that this reserved list should contain:

abs(E)	for the modulus (absolute value) of the value of the expression E
sign(E)	for the sign of the value E (+1 for $E > 0$, 0 for $E=0$, -1 for $E < 0$)
sqrt(E)	for the square root of the value of E
sin(E)	for the sine of the value of E
cos(E)	for the cosine of the value of E
arctan(E)	for the principal value of the arctangent of the value of E
ln(E)	for the natural logarithm of the value of E
exp(E)	for the exponential function of the value of E (e^E).

These functions are all understood to operate indifferently on arguments both of type real and integer. They will all yield values of type real, except for sign (E) which will have values of type integer. In a particular representation these functions may be available without explicit declarations (cf. section 5 Declarations).

3.2.4 Standard Procedures

The list of reserved identifiers is expanded to include the following:

INLIST	GET	SYSPARAM	BACKSPACE
OUTLIST	PUT	EQUIV	DUMP
INPUT	GETITEM	STRINGELEMENT	CLOCK
OUTPUT	PUTITEM	CHLENGTH	READECS
INREAL	FETCHITEM	POSITION	WRITEECS
OUTREAL	STOREITEM	THRESHOLD	MOVE
INARRAY	H LIM	INRANGE	MATMULT
OUTARRAY	V LIM	IOLTH	VINIT
INCHARACTER	H END	ERROR	VXMIT
OUTCHARACTER	V END	CHANERROR	VMONOD
GETLIST	NODATA	ARTHOFLOW	VDIAD
PUTLIST	FORMAT	PARITY	VSDIAD
FETCHLIST	TABULATION	EOF	VPROSUM
STORELIST		BAD DATA	VDOT
GETARRAY		SKIPF	BNOT
PUTARRAY		SKIPB	VBOOL
		ENDFILE	VSBOOL
		REWIND	VREL
		UNLOAD	VSREL

These procedures are described in later chapters.

Calls to all standard procedures conform to the syntax of calls to declared procedures (section 4.7.1) and in all respects are equivalent to regular procedure calls. This specifically includes the use of a standard procedure identifier as an actual parameter in a procedure call.

If a standard procedure is not needed throughout a program, its identifier may be declared to have another meaning at any level; the identifier assumes the new meaning rather than that of a standard procedure.

Since all standard procedures are contained on the system library, any object program (chapter 5) can call them without loading special libraries.

3.2.5 TRANSFER FUNCTIONS

It is understood that transfer functions between any pair of quantities and expressions may be defined. Among the standard functions it is recommended that there be one, namely,

entier(E),

which “transfers” an expression of real type to one of integer type, and assigns to it the value which is the largest integer not greater than the value of E.

3.3 ARITHMETIC EXPRESSIONS

3.3.1 SYNTAX

< adding operator > ::= + | -

< multiplying operator > ::= × | / | ÷

< primary > ::= < unsigned number > | < variable > |

< function designator > | (< arithmetic expression >)

< factor > ::= < primary > | < factor > ↑ < primary >

< term > ::= < factor > | < term > < multiplying operator > < factor >

< simple arithmetic expression > ::= < term > |

< adding operator > < term > | < simple arithmetic expression > < adding operator > < term >

< if clause > ::= if < Boolean expression > then

< arithmetic expression > ::= < simple arithmetic expression > |

< if clause > < simple arithmetic expression > else < arithmetic expression >

3.3.2 EXAMPLES

Primaries:

7.394₁₀-8
sum
w [i+2,8]
cos (y+z×3)
(a-3/y+vu↑8)

Factors:

omega
sum ↑cos(y+z×3)
7.394₁₀-8 ↑w[i+2,8] ↑ (a-3/y+vu↑8)

Terms:

U
omega×sum↑cos(y+z×3)/7.349₁₀-8↑w[i+2,8] ↑ (a-3/y+vu ↑ 8)

Simple arithmetic expression:

U-Yu+omega×sum↑cos(y+z×3)/7.349₁₀-8↑w[i+2,8] ↑
(a-3/y+vu↑8)

Arithmetic expressions:

w×u-Q(S+Cu)↑2
if q > 0 then S+3×Q/A else 2×S+3×q
if a < 0 then U+V else if a×b > 17 then U/V else if k≠y then V/U else 0
a×sin(omega×t)
0.57₁₀ 12×a[N×(N-1)/2,0]
(A×arctan(y)+Z)↑(7+Q)
if q then n-1 else n
if a < 0 then A/B else if b = 0 then B/A else z

3.3.3 SEMANTICS

An arithmetic expression is a rule for computing a numerical value. In case of simple arithmetic expression this value is obtained by executing the indicated arithmetic operations on the actual numerical values of the primaries of the expression, as explained in detail in section 3.3.4 below. The actual numerical value of a primary is obvious in the case of numbers. For variables it is the current value (assigned last in the dynamic sense), and for function designation it is the value arising from the computing rules defining the procedure (cf. section 5.4.4. Values of Function Designators) when applied to the current values of the procedure parameters given in the expression. Finally, for arithmetic expressions enclosed in parentheses the value must through a recursive analysis be expressed in terms of the values of primaries of the other three kinds.

In the more general arithmetic expressions, which include if clauses, one out of several simple arithmetic expressions is selected on the basis of the actual values of the Boolean expressions (cf. section 3.4 Boolean Expressions). This selection is made as follows: The Boolean expressions of the if clauses are evaluated one by one in sequence from left to right until one having the value true is found. The value of the arithmetic expression is then the value of the first arithmetic expression following this Boolean (the largest arithmetic expression found in this position is understood).

The construction:

else < simple arithmetic expression >

is equivalent to the construction

else if true then < simple arithmetic expression >

3.3.3 Semantics

The primaries within a simple arithmetic expression are evaluated in sequence from left to right.

In the evaluation of arithmetic expressions that include if clauses, one of several simple arithmetic expressions is selected on the basis of the actual values of the Boolean expressions (section 3.4). Such an arithmetic expression has the form: if B then E else F, where B is a Boolean expression, E is a simple arithmetic expression and F is an arithmetic expression. This selection is made as follows. The Boolean expression B is evaluated. If its value is true, the expression E is selected. If the value of B is false, the expression F is selected. The expression F can be another arithmetic expression of this form to be evaluated according to the same rule. The type of the expression is integer if E and F are both type integer; otherwise it is real.

3.3.4 OPERATORS AND TYPES

Apart from the Boolean expressions of if clauses, the constituents of simple arithmetic expressions must be of types real or integer (cf. section 5.1. Type Declarations). The meaning of the basic operators and the types of the expressions to which they lead are given by the following rules:

3.3.4.1 The operators +, -, and × have the conventional meaning (addition, subtraction, and multiplication). The type of the expression will be integer if both of the operands are of integer type, otherwise real.

3.3.4.2 The operations < term > / < factor > and < term > ÷ < factor > both denote division, to be understood as a multiplication of the term by the reciprocal of the factor with due regard to the rules of precedence (cf. section 3.3.5). Thus for example

$$a/b \times 7 / (p-q) \times v/s$$

means

$$(((a \times (b^{-1})) \times 7) \times ((p-q)^{-1})) \times v) \times (s^{-1})$$

The operator / is defined for all four combinations of types real and integer and will yield results of real type in any case. The operator ÷ is defined only for two operands both of type integer and will yield a result of type integer, mathematically defined as follows:

$$a \div b = \text{sign}(a/b) \times \text{entier}(\text{abs}(a/b))$$

(cf. sections 3.2.4 and 3.2.5).

3.3.4.2 Operators and types

When the type of an arithmetic expression cannot be determined at compile time, it is considered real. For example, the parenthesized expression in the following statement is considered real if one or both of the arithmetic expressions R and S is real:

$$P \quad \times \quad (\text{if } Q \text{ then } R \text{ else } S)$$

If both operands in a simple arithmetic expression are numbers, transformation (from type real to integer, or integer to real) and the operation itself are performed at compile time. The type of the resulting number is defined according to the number types and the particular operation involved.

If the result of an expression is assigned to a variable with a different type, the compiler generates the code to transform the result to the proper type (section 4.2.4).

When the final result is a number, type transformation is performed at compile time (as in the assignment of a simple number to a variable of a different type).

The internal representations of type real and integer values and the transformations between them are described in section 5.1.3.

3.3.4.3 The operation $\langle \text{factor} \rangle \uparrow \langle \text{primary} \rangle$ denotes exponentiation, where the factor is the base and the primary is the exponent. Thus, for example

$$2 \uparrow n \uparrow k \quad \text{means } (2^n)^k$$

while

$$2 \uparrow (n \uparrow m) \quad \text{means } 2^{(n^m)}$$

Writing i for a number of integer type, r for a number of real type, and a for a number of either integer or real type, the result is given by the following rules:

$a \uparrow i$ If $i > 0$, $a \times a \times \dots \times a$ (i times), of the same type as a.

If $i = 0$, if $a \neq 0, 1$, of the same type as a.

if $a = 0$, undefined.

If $i < 0$, if $a \neq 0, 1/(a \times a \times \dots \times a)$ (the denominator has $-i$ factors), of type real.

if $a = 0$, undefined.

$a \uparrow r$ If $i > 0$, $\exp(r \times \ln(a))$, of type real.

If $a = 0$, if $r > 0, 0.0$, of type real.

if $r \leq 0$, undefined.

If $a < 0$, always undefined.

3.3.4.3 The rule for evaluating an expression of the form $a \uparrow i$ or $a \uparrow r$ is the same as defined above except when a is of type integer and i is a positive integer. In this case, if i is signed, the result is of type real; if i is unsigned, the result is of type integer. The Revised Report calls for a result of type integer in both cases.

Whenever a result is specified in paragraph 3.3.4.3 as undefined, an error message is issued by ALGOL 4.

3.3.5 PRECEDENCE OF OPERATORS

The sequence of operations within one expression is generally from left to right, with the following additional rules:

3.3.5.1 According to the syntax given in section 3.3.1 the following rules of precedence hold:

first: \uparrow

second: \times / \div

third: $+ -$

3.3.5.2 The expression between a left parenthesis and the matching right parenthesis is evaluated by itself and this value is used in subsequent calculations. Consequently the desired order of execution of operations within an expression can always be arranged by appropriate positioning of parentheses.

3.3.6 ARITHMETICS OF REAL QUANTITIES

Numbers and variables of type real must be interpreted in the sense of numerical analysis, i.e. as entities defined inherently with only a finite accuracy. Similarly, the possibility of the occurrence of a finite deviation from the mathematically defined result in any arithmetic expression is explicitly understood. No exact arithmetic will be specified, however, and it is indeed understood that different hardware representations may evaluate arithmetic expressions differently. The control of the possible consequences of such differences must be carried out by methods of numerical analysis. This control must be considered a part of the process to be described, and will therefore be expressed in terms of the language itself.

3.4 BOOLEAN EXPRESSIONS

3.4.1 SYNTAX

$\langle \text{relational operator} \rangle ::= < | \leq | = | \geq | > | \neq$

$\langle \text{relation} \rangle ::= \langle \text{simple arithmetic expression} \rangle \langle \text{relational operator} \rangle \langle \text{simple arithmetic expression} \rangle$

$\langle \text{Boolean primary} \rangle ::= \langle \text{logical value} \rangle \mid \langle \text{variable} \rangle \mid$
 $\langle \text{function designator} \rangle \mid \langle \text{relation} \rangle \mid (\langle \text{Boolean expression} \rangle)$
 $\langle \text{Boolean secondary} \rangle ::= \langle \text{Boolean primary} \rangle \mid \neg \langle \text{Boolean primary} \rangle$
 $\langle \text{Boolean factor} \rangle ::= \langle \text{Boolean secondary} \rangle \mid$
 $\langle \text{Boolean factor} \rangle \wedge \langle \text{Boolean secondary} \rangle$
 $\langle \text{Boolean term} \rangle ::= \langle \text{Boolean factor} \rangle \mid \langle \text{Boolean term} \rangle \vee \langle \text{Boolean factor} \rangle$
 $\langle \text{implication} \rangle ::= \langle \text{Boolean term} \rangle \mid \langle \text{implication} \rangle \supset \langle \text{Boolean term} \rangle$
 $\langle \text{simple Boolean} \rangle ::= \langle \text{implication} \rangle \mid$
 $\langle \text{simple Boolean} \rangle \equiv \langle \text{implication} \rangle$
 $\langle \text{Boolean expression} \rangle ::= \langle \text{simple Boolean} \rangle \mid$
 $\langle \text{if clause} \rangle \langle \text{simple Boolean} \rangle \underline{\text{else}} \langle \text{Boolean expression} \rangle$

3.4.2 EXAMPLES

$x = -2$
 $Y > V \vee z < q$
 $a + b > -5 \wedge z - d > q \uparrow 2$
 $p \wedge q \vee x \neq y$
 $g \equiv \neg a \wedge b \wedge \neg c \vee d \vee e \supset \neg f$
 $\underline{\text{if}} \ k < 1 \ \underline{\text{then}} \ s > w \ \underline{\text{else}} \ h \leq c$
 $\underline{\text{if}} \ \underline{\text{if}} \ a \ \underline{\text{then}} \ b \ \underline{\text{else}} \ c \ \underline{\text{then}} \ d \ \underline{\text{else}} \ f \ \underline{\text{then}} \ g \ \underline{\text{else}} \ h < k$

3.4.3 SEMANTICS

A Boolean expression is a rule for computing a logical value. The principles of evaluation are entirely analogous to those given for arithmetic expressions in section 3.3.3.

3.4.4 TYPES

Variables and function designators entered as Boolean primaries must be declared Boolean (cf. section 5.1. Type Declarations and section 5.4.4 Values of Function Designators).

3.4.5 THE OPERATORS

Relations take on the value true whenever the corresponding relation is satisfied for the expressions involved, otherwise false.

The meaning of the logical operators \neg (not), \wedge (and), \vee (or), \supset (implies), and \equiv (equivalent), is given by the following function table.

b1	<u>false</u>	<u>false</u>	<u>true</u>	<u>true</u>
b2	<u>false</u>	<u>true</u>	<u>false</u>	<u>true</u>
$\neg b1$	<u>true</u>	<u>true</u>	<u>false</u>	<u>false</u>
$b1 \wedge b2$	<u>false</u>	<u>false</u>	<u>false</u>	<u>true</u>
$b1 \vee b2$	<u>false</u>	<u>true</u>	<u>true</u>	<u>true</u>
$b1 \supset b2$	<u>true</u>	<u>true</u>	<u>false</u>	<u>true</u>
$b1 \equiv b2$	<u>true</u>	<u>false</u>	<u>false</u>	<u>true</u>

3.4.6 PRECEDENCE OF OPERATORS

The sequence of operations within one expression is generally from left to right, with the following additional rules:

3.4.6.1 According to the syntax given in section 3.4.1 the following rules of precedence hold:

first: arithmetic expressions according to section 3.3.5

second: $< \leq = \geq > \neq$

third: \neg

fourth: \wedge

fifth: \vee

sixth: \supset

seventh: \equiv

3.4.6.2 The use of parentheses will be interpreted in the sense given in section 3.3.5.2.

3.5 DESIGNATIONAL EXPRESSIONS

3.5.1 SYNTAX

$\langle \text{label} \rangle ::= \langle \text{identifier} \rangle \mid \langle \text{unsigned integer} \rangle$

$\langle \text{switch identifier} \rangle ::= \langle \text{identifier} \rangle$

$\langle \text{switch designator} \rangle ::= \langle \text{switch identifier} \rangle [\langle \text{subscript expression} \rangle]$

$\langle \text{simple designational expression} \rangle ::= \langle \text{label} \rangle \mid \langle \text{switch designator} \rangle \mid$
 $(\langle \text{designational expression} \rangle)$

$\langle \text{designational expression} \rangle ::= \langle \text{simple designational expression} \rangle \mid$
 $\langle \text{if clause} \rangle \langle \text{simple designational expression} \rangle \underline{\text{else}} \langle \text{designational expression} \rangle$

3.5.2 EXAMPLES

```
17
p9
Choose [n-1]
Town [if y < 0 then N else N+1]
if Ab < c then 17 else q [if w ≤ 0 then 2 else n]
```

3.5.3 SEMANTICS

A designational expression is a rule for obtaining a label of a statement (cf. section 4. Statements). Again the principle of the evaluation is entirely analogous to that of arithmetic expressions (section 3.3.3). In the general case the Boolean expressions of the if clauses will select a simple designational expression. If this is a label the desired result is already found. A switch designator refers to the corresponding switch declaration (cf. section 5.3 Switch Declarations) and by the actual numerical value of its subscript expression selects one of the designational expressions listed in the switch declaration by counting these from left to right. Since the designational expression thus selected may again be a switch designator this evaluation is obviously a recursive process.

3.5.4 THE SUBSCRIPT EXPRESSION

The evaluation of the subscript expression is analogous to that of subscripted variables (cf. section 3.1.4.2). The value of a switch designator is defined only if the subscript expression assumes one of the positive values 1,2,3,...,n, where n is the number of entries in the switch list.

3.5.5 UNSIGNED INTEGERS AS LABELS

Unsigned integers used as labels have the property that leading zeros do not affect their meaning, e.g. 00217 denotes the same label as 217.

3.5.5 Unsigned Integers as Labels

Integer labels are not permitted. Therefore, the definition of $\langle \text{label} \rangle$ is replaced by:

$$\langle \text{label} \rangle ::= \langle \text{identifier} \rangle$$

Note also that the first example in 3.5.2 is invalid for ALGOL 4.

4. Statements

The units of operation within the language are called statements. They will normally be executed consecutively as written. However, this sequence of operations may be broken by go to statements, which define their successor explicitly, and shortened by conditional statements, which may cause certain statements to be skipped.

In order to make it possible to define a specific dynamic succession, statements may be provided with labels.

Since sequences of statements may be grouped together into compound statements and blocks the definition of statement must necessarily be recursive. Also since declarations, described in section 5, enter fundamentally into the syntactic structure, the syntactic definition of statements must suppose declarations to be already defined.

4.1 COMPOUND STATEMENTS AND BLOCKS

4.1.1 SYNTAX

$$\langle \text{unlabelled basic statement} \rangle ::= \langle \text{assignment statement} \rangle \mid$$
$$\langle \text{go to statement} \rangle \mid \langle \text{dummy statement} \rangle \mid \langle \text{procedure statement} \rangle$$
$$\langle \text{basic statement} \rangle ::= \langle \text{unlabelled basic statement} \rangle \mid \langle \text{label} \rangle : \langle \text{basic statement} \rangle$$
$$\langle \text{unconditional statement} \rangle ::= \langle \text{basic statement} \rangle \mid$$
$$\langle \text{compound statement} \rangle \mid \langle \text{block} \rangle$$
$$\langle \text{statement} \rangle ::= \langle \text{unconditional statement} \rangle \mid$$
$$\langle \text{conditional statement} \rangle \mid \langle \text{for statement} \rangle$$
$$\langle \text{compound tail} \rangle ::= \langle \text{statement} \rangle \underline{\text{end}} \mid \langle \text{statement} \rangle ; \langle \text{compound tail} \rangle$$

$\langle \text{block head} \rangle ::= \underline{\text{begin}} \langle \text{declaration} \rangle \mid \langle \text{block head} \rangle ; \langle \text{declaration} \rangle$

$\langle \text{unlabelled compound} \rangle ::= \underline{\text{begin}} \langle \text{compound tail} \rangle$

$\langle \text{unlabelled block} \rangle ::= \langle \text{block head} \rangle ; \langle \text{compound tail} \rangle$

$\langle \text{compound statement} \rangle ::= \langle \text{unlabelled compound} \rangle \mid$

$\quad \langle \text{label} \rangle : \langle \text{compound statement} \rangle$

$\langle \text{block} \rangle ::= \langle \text{unlabelled block} \rangle \mid \langle \text{label} \rangle : \langle \text{block} \rangle$

$\langle \text{program} \rangle ::= \langle \text{block} \rangle \mid \langle \text{compound statement} \rangle$

This syntax may be illustrated as follows: Denoting arbitrary statements, declarations, and labels, by the letters S, D, and L, respectively, the basic syntactic units take the forms:

Compound statement:

L: L: . begin S;S; . . S;S end

Block:

L: L: . . begin D;D; . . D;S;S; . . S;

S end

It should be kept in mind that each of the statement S may again be a complete compound statement or block.

4.1.2 EXAMPLES

Basic Statements:

a:=p+q
go to Naples
START:CONTINUE:W:=7.993

Compound Statement:

begin x:=0;for y:=1 step 1 until n do
 x:=x+A[y];
 if x > q then go to STOP else if x > w-2 then go to S;
 Aw:St:W:=x+bob end

Block:

```
Q: begin integer i,k; real w;  
  for i : = 1 step 1 until m do  
    for k : = i+1 step 1 until m do  
      begin w : = A [i,k] ;  
        A [i,k] := A [k,i] ;  
        A [k,i] := w end for i and k  
      end block Q
```

4.1.3 SEMANTICS

Every block automatically introduces a new level of nomenclature. This is realized as follows: Any identifier occurring within the block may through a suitable declaration (cf. section 5. Declarations) be specified to be local to the block in question. This means (a) that the entity represented by this identifier inside the block has no existence outside it, and (b) that any entity represented by this identifier outside the block is completely inaccessible inside the block.

Identifiers (except those representing labels) occurring within a block and not being declared to this block will be non-local to it, i.e., will represent the same entity inside the block and in the level immediately outside it. A label separated by a colon from a statement, i.e., labelling that statement, behaves as though declared in the head of the smallest embracing block, i.e., the smallest block whose brackets begin and end enclose that statement. In this context a procedure body must be considered as if it were enclosed by begin and end and treated as a block. Since a statement of a block may again itself be a block the concepts local and nonlocal to a block must be understood recursively. Thus an identifier, which is nonlocal to a block A, may or may not be nonlocal to the block B in which A is one statement.

4.1.3 Semantics

Program labels are not allowed in ALGOL 4. The maximum number of blocks permitted in a single compilation is 253. The maximum static nesting permitted for blocks is 63. A procedure body counts as a block in this context. Blocks may be overlayed; for a description of this facility, see chapter 10.

4.2 ASSIGNMENT STATEMENTS

4.2.1 SYNTAX

< left part > ::= < variable > : = | < procedure identifier > : =

< left part list > ::= < left part > | < left part list > < left part >

< assignment statement > ::= < left part list > < arithmetic expression > |

< left part list > < Boolean expression >

4.2.2 EXAMPLES

```
s := p [0] := n := n+1+s  
n := n+1  
A := B/C-v-qXS  
S [v,k+2] := 3-arctan(sXzeta)  
V := Q > Y ^ Z
```

4.2.3 SEMANTICS

Assignment statements serve for assigning the value of an expression to one or several variables or procedure identifiers. Assignment to a procedure identifier may only occur within the body of a procedure defining the value of a function designator (cf. section 5.4.4). The process will in the general case be understood to take place in three steps as follows:

4.2.3 Semantics

To be precise, the second sentence in the preceding paragraph should read: "Assignment to a procedure identifier may occur only within the body of a procedure defining a function designator of the same name."

4.2.3.1 Any subscript expressions occurring in the left part variables are evaluated in sequence from left to right.

4.2.3.1 This section is ambiguous since subscripts may contain subscripted variables. It is amended to:

The subscript expressions of any subscripted variables occurring as left part variables are evaluated in sequence from left to right.

If the assignment statement is in a procedure body, any formal parameters in the left part list are replaced by actual parameters according to section 4.7.3.2 before the evaluation of subscript expressions takes place.

4.2.3.2 The expression of the statement is evaluated.

4.2.3.3 The value of the expression is assigned to all the left part variables, with any subscript expressions having values as evaluated in step 4.2.3.1.

4.2.4 TYPES

The type associated with all variables and procedure identifiers of a left part list must be the same. If this type is Boolean the expression must likewise be Boolean. If the type is real or integer, the expression must be arithmetic. If the type of the arithmetic expression differs from that associated with the variables and procedure identifiers, appropriate transfer functions are understood to be automatically invoked. For transfer from real to integer type, the transfer function is understood to yield a result equivalent to

entier(E+0.5)

where E is the value of the expression. The type associated with a procedure identifier is given by the declarator which appears as the first symbol of the corresponding procedure declaration (cf. section 5.4.4).

4.2.4 Types

If the type of an arithmetic expression (Section 3.3.4) is different from that of the variable or procedure identifier to which it is assigned, the compiler generates the code to perform the transformation from one type to the other.

The internal representations of real and integer values and the transformations between them are described in Section 5.1.3.

4.3 GO TO STATEMENTS

4.3.1 SYNTAX

< go to statement > ::= go to < designational expression >

4.3.2 EXAMPLES

go to 8
go to exit [n+1]
go to Town [if y < 0 then N else N+1]
go to if Ab < c then 17 else q [if w < 0 then 2 else n]

4.3.3 SEMANTICS

A go to statement interrupts the normal sequence of operations, defined by the write-up of statements, by defining its successor explicitly by the value of a designational expression. Thus the next statement to be executed will be the one having this value as its label.

4.3.4 RESTRICTION

Since labels are inherently local, no go to statement can lead from outside into a block. A go to statement may, however, lead from outside into a compound statement.

4.3.5 GO TO AN UNDEFINED SWITCH DESIGNATOR

A go to statement is equivalent to a dummy statement if the designational expression is a switch designator whose value is undefined.

4.3.5 Go To an Undefined Switch Designator.

If the designational expression of a go to statement is a switch designator whose value is undefined when an attempt is made to execute the go to statement, the object program terminates abnormally.

4.4 DUMMY STATEMENTS

4.4.1 SYNTAX

< dummy statement > ::= < empty >

4.4.2 EXAMPLES

**L:
begin. . . ;John:end**

4.4.3 SEMANTICS

A dummy statement executes no operation. It may serve to place a label.

4.5 CONDITIONAL STATEMENTS

4.5.1 SYNTAX

< if clause > ::= if < Boolean expression > then

< unconditional statement > ::= < basic statement > |

< compound statement > | < block >

< if statement > ::= < if clause > < unconditional statement >

< conditional statement > ::= < if statement > | < if statement > else < statement > |

< if clause > < for statement > | < label > : < conditional statement >

4.5.2 EXAMPLES

```

if x > 0 then n := n+1
if v > u then V: q:=n+m else go to R
if s < 0 ∨ P ≤ Q then AA: begin if q < v then a := v/s
    else y := 2×a end
    else if v > s then a := v-q else if v > s-1
        then go to S

```

4.5.3 SEMANTICS

Conditional statements cause certain statements to be executed or skipped depending on the running values of specified Boolean expressions.

4.5.3.1 If statement. The unconditional statement of an if statement will be executed if the Boolean expression of the if clause is true. Otherwise it will be skipped and the operation will be continued with the next statement.

4.5.3.2 Conditional statement. According to the syntax two different forms of conditional statements are possible. These may be illustrated as follows:

if B1 then S1 else if B2 then S2 else S3; S4

and

if B1 then S1 else if B2 then S2 else if B3 then S3; S4

Here B1 to B3 are Boolean expressions, while S1 to S3 are unconditional statements. S4 is the statement following the complete conditional statement.

The execution of a conditional statement may be described as follows: The Boolean expression of the if clauses are evaluated one after the other in sequence from left to right until one yielding the value true is found. Then the unconditional statement following this Boolean is executed. Unless this statement defines its successor explicitly the next statement to be executed will be S4, i.e., the statement following the complete conditional statement. Thus the effect of the delimiter else may be described by saying that it defines the successor of the statement it follows to be the statement following the complete conditional statement.

The construction

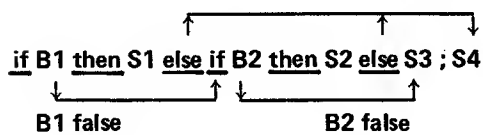
else < unconditional statement >

is equivalent to

else if true then < unconditional statement >

If none of the Boolean expressions of the if clause, is true, the effect of the whole conditional statement will be equivalent to that of a dummy statement.

For further explanation the following picture may be useful:



4.5.3.2 Conditional statements. According to the syntax, three forms of unlabelled conditional statements are possible.

These may be illustrated as follows:

if B then S
if B then S else T
if B then U

Here B is a Boolean expression, S is an unconditional statement, T is any statement, and U is a for-statement.

The execution of a conditional statement may be described as follows:

The Boolean expression B is evaluated. If its value is true, the statement S or U is executed. If its value is false and if the conditional statement has the second form, the statement T is executed (this statement can be another conditional statement, to be interpreted according to the same rule).

4.5.4 GO TO INTO A CONDITIONAL STATEMENT

The effect of a go to statement leading into a conditional statement follows directly from the above explanation of the effect of else.

4.5.4 Go to into a conditional statement

If a go to statement refers to a label within S or U, the effect is the same as if the remainder of the conditional statement ('if B then' and in the second case also 'else T') were not present.

4.6 FOR STATEMENTS

4.6.1 SYNTAX

$\langle \text{for list element} \rangle ::= \langle \text{arithmetic expression} \rangle \mid$

$\langle \text{arithmetic expression} \rangle \underline{\text{step}} \langle \text{arithmetic expression} \rangle \underline{\text{until}}$

$\langle \text{arithmetic expression} \rangle \mid \langle \text{arithmetic expression} \rangle \underline{\text{while}} \langle \text{Boolean expression} \rangle$

$\langle \text{for list} \rangle ::= \langle \text{for list element} \rangle \mid \langle \text{for list} \rangle , \langle \text{for list element} \rangle$

$\langle \text{for clause} \rangle ::= \underline{\text{for}} \langle \text{variable} \rangle : = \langle \text{for list} \rangle \underline{\text{do}}$

$\langle \text{for statement} \rangle ::= \langle \text{for clause} \rangle \langle \text{statement} \rangle \mid$

$\langle \text{label} \rangle : \langle \text{for statement} \rangle$

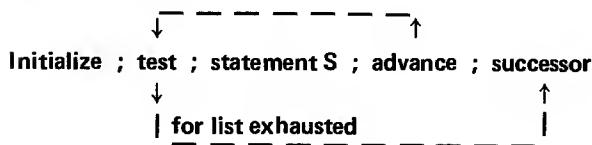
4.6.2 EXAMPLES

```

for q := 1 step s until n do A [q] := B [q]
for k := 1, V1X2 while V1 < N do
  for j := 1+G,L,1 step 1 until N,C+D do
    A [k,j] := B [k,j]
  
```

4.6.3 SEMANTICS

A for clause causes the statement S which it precedes to be repeatedly executed zero or more times. In addition, it performs a sequence of assignments to its controlled variable. The process may be visualized by means of the following picture:



In this picture the word initialize means: perform the first assignment of the for clause. Advance means: perform the next assignment of the for clause. Test determines if the last assignment has been done. If so, the execution continues with the successor of the for statement. If not, the statement following the for clause is executed.

4.6.3 Semantics

In a for-statement, if the controlled variable is subscripted, the same array element is used as the control variable throughout the execution of the for-statement, regardless of any changes that might occur to the value of the subscript expressions during its execution. The element used is the one referenced by the value of the subscript expressions on entry to the for-statement.

4.6.4 THE FOR LIST ELEMENTS

The for list gives a rule for obtaining the values which are consecutively assigned to the controlled variable. This sequence of values is obtained from the for list elements by taking these one by one in the order in which they are written. The sequence of values generated by each of the three species of for list elements and the corresponding execution of the statement S are given by the following rules:

4.6.4.1 Arithmetic expression. This element gives rise to one value, namely the value of the given arithmetic expression as calculated immediately before the corresponding execution of the statement S.

4.6.4.2 Step-until-element. An element of the form A step B until C, where A,B and C are arithmetic expressions, gives rise to an execution which may be described most concisely in terms of additional ALGOL statements as follows:

```

    V := A ;
L1 : if (V-C) × sign(B) > 0 then go to element exhausted;
    statement S ;
    V := V+B ;
    go to L1 ;

```

where V is the controlled variable of the for clause and *element exhausted* points to the evaluation according to the next element in the for list, or if the step-until-element is the last of the list, to the next statement in the program.

4.6.4.2 Step-until-element. The execution governed by a step-until element is as described so far as possible side-effects are concerned. However, determination of the location of V is performed only once, at the entry to the for clause. This can result in side-effects in the case where V is a subscripted variable containing a call to a procedure which modifies the value of V.

Secondly, the step element is evaluated only once for each execution of the step expression. That value is used to increment the for variable and to multiply with when testing the limit.

4.6.4.3 While element. The execution governed by a for list element of the form E while F, where E is an arithmetic and F a Boolean expression, is most concisely described in terms of additional ALGOL statements as follows:

```

L3: V := E ;
    if  $\neg$  F then go to element exhausted ;
    Statements S ;
    go to L3 ;

```

where the notation is the same as in 4.6.4.2. above.

4.6.4.3 While-element. The execution governed by a while element is as described so far as possible side-effects are concerned. However, the determination of the location V is performed only once, at the entry to the for clause. This can result in side-effects in the case where V is a subscripted variable containing a call to a procedure which modifies the value of V.

4.6.5 THE VALUE OF THE CONTROLLED VARIABLE UPON EXIT

Upon exit out of the statement S (supposed to be compound) through a go to statement the value of the controlled variable will be the same as it was immediately preceding the execution of the go to statement.

If the exit is due to exhaustion of the for list, on the other hand, the value of the controlled variable is undefined after the exit.

4.6.5 The value of the controlled variable upon exit

In the case of step-until and while-elements, the value of the controlled variable on exit from a for-statement through exhaustion of the for list is the value causing the failure of the test. In the case of a simple for element, the value of the controlled variable on exit is the same as it was at the end of the final execution of the statement S.

4.6.6 GO TO LEADING INTO A FOR STATEMENT

The effect of a go to statement, outside a for statement, which refers to a label within the for statement, is undefined.

4.6.6 Go to leading into a for-statement

Any occurrence of a label in a designational expression outside the for-statement which contains that label and the statement it labels causes a warning message at compile time. Any go to statement outside a for-statement referring to a label within the for-statement causes this warning. However, the legitimate case of a go to statement in a procedure called from within a for-statement which refers to a label within that for-statement via an actual parameter of the procedure call does not cause a warning. It is permissible to specify a transfer from within a procedure to a label in the block calling the procedure.

The effect of executing an object program containing a go to statement leading into a for-statement is undefined despite the warning.

4.7 PROCEDURE STATEMENTS

4.7.1 Syntax

< actual parameter > ::= < string > | < expression > | < array identifier > |

< switch identifier > | < procedure identifier >

< letter string > ::= < letter > | < letter string > < letter >

< parameter delimiter > ::= , |) < letter string > : (

< actual parameter list > ::= < actual parameter > | < actual parameter list > < parameter delimiter > < actual parameter >

< actual parameter part > ::= < empty > |

(< actual parameter list >)

< procedure statement > ::= < procedure identifier > < actual parameter part >

4.7.2 EXAMPLES

```
Spur(A)Order: (7)Result to: (V)
Transpose (W,v+1)
Absmax (A,N,M,Yy,I,K)
Innerproduct (A [t,P,u] ,B [P] ,10,P,Y)
```

These examples correspond to examples given in Section 5.4.2.

4.7.3 SEMANTICS

A procedure statement serves to invoke (call for) the execution of a procedure body (cf. Section 5.4. Procedure Declarations). Where the procedure body is a statement written in ALGOL the effect of this execution will be equivalent to the effect of performing the following operations on the program at the time of execution of the procedure statement:

4.7.3.1 Value assignment (call by value)

All formal parameters quoted in the value part of the procedure declaration heading are assigned the values (cf. section 2.8. Values and Types) of the corresponding actual parameters, these assignments being considered as being performed explicitly before entering the procedure body. The effect is as though an additional block embracing the procedure body were created in which these assignments were made to variables local to this fictitious block with types as given in the corresponding specifications (cf. Section 5.4.5). As a consequence, variables called by value are to be considered as non local to the body of the procedure, but local to the fictitious block (cf. section 5.4.3).

4.7.3.1 Value assignment (call by value)

The actual parameters corresponding to formal parameters called by value are evaluated first from left to right for non-arrays, then from left to right for arrays, in sequence as they appear in the actual parameter list.

4.7.3.2 Name replacement (call by name)

Any formal parameter not quoted in the value list is replaced, throughout the procedure body by the corresponding actual parameter, after enclosing this latter in parentheses wherever syntactically possible. Possible conflicts between identifiers inserted through this process and other identifiers already present within the procedure body will be avoided by suitable systematic changes of the formal or local identifiers involved.

4.7.3.3 Body replacement and execution

Finally the procedure body, modified as above, is inserted in place of the procedure statement and executed. If the procedure is called from a place outside the scope of any nonlocal quantity of the procedure body the conflicts between the identifiers inserted through this process of body replacement and the identifiers whose declarations are valid at the place of the procedure statement or function designator will be avoided through suitable systematic changes of the latter identifiers.

4.7.4 ACTUAL-FORMAL CORRESPONDENCE

The correspondence between the actual parameters of the procedure statement and the formal parameters of the procedure heading is established as follows: the actual parameter list of the procedure statement must have the same number of entries as the formal parameter list of the procedure declaration heading. The correspondence is obtained by taking the entries of these two lists in the same order.

4.7.5 RESTRICTIONS

For a procedure statement to be defined it is evidently necessary that the operations on the procedure body defined in sections 4.7.3.1. and 4.7.3.2. lead to a correct ALGOL statement. This imposes the restriction on any procedure statement that the kind and type of each actual parameter be compatible with the kind and type of the corresponding formal parameter. Some important particular cases of this general rule are the following:

4.7.5.1 If a string is supplied as an actual parameter in a procedure statement or function designator, whose defining procedure body is an ALGOL-60 statement (as opposed to non-ALGOL code, cf. Section 4.7.8.), then this string can only be used within the procedure body as an actual parameter in further procedure calls. Ultimately it can only be used by a procedure body expressed in non-ALGOL code.

4.7.5.2 A formal parameter which occurs as a left part variable in an assignment statement within the procedure body and which is not called by value can only correspond to an actual parameter which is variable (special case of expression).

4.7.5.3 A formal parameter which is used within the procedure body as an array identifier can only correspond to an actual parameter which is an array identifier of an array of the same dimensions. In addition, if the formal parameter is called by value the local array created during the call will have the same subscript bounds as the actual array.

4.7.5.4 A formal parameter which is called by value cannot in general correspond to a switch identifier or a procedure identifier or a string, because these latter do not possess values (the exception is the procedure identifier of a procedure declaration which has an empty formal parameter part (cf. section 5.4.1) and which defines the value of a function designator (cf. section 5.4.4). This procedure identifier is in itself a complete expression).

4.7.5.5 Any formal parameter may have restrictions on the type of the corresponding actual parameter associated with it (these restrictions may, or may not, be given through specifications in the procedure heading). In the procedure statement such restrictions must evidently be observed.

4.7.5 Restrictions

A maximum of 63 formal parameters can be included in a procedure declaration (section 5.4.3); therefore, a maximum of 63 actual parameters can be included in a procedure call.

A label cannot be specified by value. The types of real and integer actual parameters should correspond to the types of the formal parameters. However, the control statement option X (chapter 6, section 6.1.2) permits mixed actual-formal correspondence for real and integer parameters at the possible expense of object time efficiency.

4.7.6 DELETED

4.7.7 PARAMETER DELIMITERS

All parameter delimiters are understood to be equivalent. No correspondence between the parameter delimiters used in a procedure statement and those used in the procedure heading is expected beyond their number being the same. Thus the information conveyed by using the elaborate ones is entirely optional.

4.7.8 PROCEDURE BODY EXPRESSED IN CODE

The restrictions imposed on a procedure statement calling a procedure having its body expressed in non-ALGOL code evidently can only be derived from the characteristics of the code used and the intent of the user and thus fall outside the scope of the reference language.

4.7.8 Procedure body expressed in code

The symbols code, algol and fortran are included to permit reference to procedures which are compiled separately from the program or procedure in which they are referenced (section 5.4.6.).

5. Declarations

Declarations serve to define certain properties of the quantities used in the program, and to associate them with identifiers. A declaration of an identifier is valid for one block. Outside this block the particular identifier may be used for other purposes (cf. section 4.1.3.).

Dynamically this implies the following: at the time of an entry into a block (through the begin, since the labels inside are local and therefore inaccessible from outside) all identifiers declared for the block assume the significance implied by the nature of the declarations given. If these identifiers had already been defined by other declarations outside, they are for the time being given a new significance. Identifiers which are not declared for the block, on the other hand, retain their old meaning.

At the time of an exit from the block (through end, or by a go to statement) all identifiers which are declared for the block lose their local significance.

A declaration may be marked with the additional declarator own. This has the following effect: upon a re-entry into the block, the values of own quantities will be unchanged from their values at the last exit, while the values of declared variables which are not marked as own are undefined. Apart from labels and formal parameters of procedure declarations and with the possible exception of those for standard functions (cf. sections 3.2.4 and 3.2.5), all identifiers of a program must be declared. No identifier may be declared more than once in any one block head.

Syntax

< declaration > ::= < type declaration > | < array declaration > | < switch declaration > | < procedure declaration >

5. Declarations

The difference between own and non-own quantities is the following:

Non-own variables are attached to the block in which they are local both with regard to the meaning of their identifiers and with regard to the existence of their values. Each entry into a block, whether recursive or not, will bring a new set of the non-own quantities of that block into existence, the values of which are initially undefined. On exit from a block all the non-own quantities created at the corresponding entry are lost, with respect both to their identifiers and to their values.

Own quantities, on the other hand, are local only with respect to the accessibility of their identifiers. Where the existence of their values is concerned they behave more closely as though declared in the outermost block of the program. Thus every entry into a block, whether recursive or not, will make the same set of values of the own quantities accessible. On exit from a block the values of the own quantities of the block are preserved, but remain inaccessible until re-entry is made to a place within the scope of the identifiers.

Each own variable and each subscripted variable corresponding to an own array is initialized to one of the following values, according to type (that is, it is given that value on its creation by first entry into the block in which it is declared):

Type	Initial Values
integer	0
real	0.0
Boolean	<u>false</u>

Certain identifiers have meaning without explicit declaration. These are the standard procedures of sections 3.2.4 and 3.2.5 which behave as though they were declared in a block embracing the entire program (including the fictitious block supplied for labels described in section 4.1.3) and any implicit outer blocks. All other identifiers of a program must be declared.

Further identifiers can be added by implicit outer blocks (see chapter 4).

5.1 TYPE DECLARATIONS

5.1.1 SYNTAX

$\langle \text{type list} \rangle = \langle \text{simple variable} \rangle \mid \langle \text{simple variable} \rangle , \langle \text{type list} \rangle$

$\langle \text{type} \rangle ::= \underline{\text{real}} \mid \underline{\text{integer}} \mid \underline{\text{Boolean}}$

$\langle \text{local or own type} \rangle ::= \langle \text{type} \rangle \mid \underline{\text{own}} \langle \text{type} \rangle$

$\langle \text{type declaration} \rangle ::= \langle \text{local or own type} \rangle \langle \text{type list} \rangle$

5.1.2 EXAMPLES

integer p,q,s
own Boolean Acryl,n

5.1.3 SEMANTICS

Type declarations serve to declare certain identifiers to represent simple variables of a given type. Real declared variables may only assume positive or negative values including zero. Integer declared variables may only assume positive and negative integral values including zero. Boolean declared variables may only assume the value true and false.

In arithmetic expressions any position which can be occupied by a real declared variable may be occupied by an integer declared variable.

For the semantics of own, see the fourth paragraph of Section 5 above.

5.1.3 Semantics

Variables of type real and integer are represented internally in 60-bit floating-point form, with a 48-bit coefficient, sign bit, and 11-bit biased exponent. All values are normalized and the complete range of absolute values is

$$(2 \uparrow 47) * 2 \uparrow (-1022) \leq \text{value} \leq (2 \uparrow 48 - 1) * 2 \uparrow 1022$$

which is approximately

$$1.6_{10} \cdot 10^{-294} \leq \text{value} \leq 1.2_{10} \cdot 10^{322}$$

The largest integer, N, which can be represented in ALGOL 4 such that the representation of $N-1 \neq N$ and the representation of $N+1 = N$ is

$$2 \uparrow 48 = 281\,474\,976\,710\,656$$

The smallest number, P, which can be distinguished when added to or subtracted from 1.0 is $2 \uparrow -47$ which is approximately $7.1_{10} \cdot 10^{-15}$.

Real and integer values with up to 15 significant decimal digits can be represented. However, the accuracy of the fifteenth digit cannot be guaranteed.

A real or integer zero is represented by a word of 60 zero bits.

Standard procedures THRESHOLD and INRANGE are provided for setting and testing limiting numeric values (chapter 3, section 3.5).

The machine provides representation for both \pm infinite and \pm indefinite values and the use of such values as operands in arithmetic operations produces run time error conditions called mode errors.

Real and integer numbers (section 2.5.4) have the same range of values and are represented in the same form as real and integer variables. In the evaluation of arithmetic expressions (section 3.3.4) and their assignment to variables of different type (section 4.2.4), conversion from real to integer is performed by closed subroutines at compile time and in-line code at object time. (No conversion is required from integer to real because of their identical internal representations.)

This conversion selects from the real value an integer value according to the rule:

$$\text{ENTIER}(\text{real value} + 0.5)$$

Variables of type Boolean are represented in 60-bit fixed-point form; only the high order bit is significant:

true ::= high-order bit = 1
false ::= high-order bit = 0

The values of own variables are global to the whole program because of their assigned position in the object program stack. They are, however, accessible only in the block in which they are declared, in the same way as any other declared variable.

5.2 ARRAY DECLARATIONS

5.2.1 SYNTAX

< lower bound > ::= < arithmetic expression >

< upper bound > ::= < arithmetic expression >

< bound pair > ::= < lower bound > : < upper bound >

< bound pair list > ::= < bound pair > | < bound pair list > , < bound pair >

< array segment > ::= < array identifier > [< bound pair list >] | < array identifier > , < array segment >

< array list > ::= < array segment > | < array list > , < array segment >

< array declaration > ::= array < array list > | < local or own type > array < array list >

5.2.2 EXAMPLES

array a,b,c [7:n,2:m] ,s [-2: 10]
own integer array A [if c < 0 then 2 else 1 : 20]
real array q [-7:-1]

5.2.3 SEMANTICS

An array declaration declares one or several identifiers to represent multidimensional arrays of subscripted variables and gives the dimensions of the arrays, the bounds of the subscripts and the type of the variables.

5.2.3.1 Subscript bounds. The subscript bounds for any array are given in the first subscript bracket following the identifier of this array in the form of a bound pair list. Each item of this list gives the lower and upper bound of a subscript in the form of two arithmetic expressions separated by the delimiter : The bound pair list gives the bounds of all subscripts taken in order from left to right.

5.2.3.2 Dimensions. The dimensions are given as the number of entries in the bound pair lists.

5.2.3.3 Types. All arrays declared in one declaration are of the same quoted type. If no type declarator is given, the type real is understood.

5.2.3 Semantics

Normally, an array is stored in central memory (CM) or small core memory (SCM). The user can allocate an array to extended core storage (ECS) or large core memory (LCM) through the S option of the ALGOL control statement (chapter 6). The user may declare an array to be virtual. A virtual array can be referenced only in its entirety and can be used only as a parameter to a procedure. (See chapters 6 and 11, and the MOVE procedure in section 3.5, chapter 3).

The bound pairs in a bound pair list are evaluated in sequence from left to right. In each bound pair, the lower bound is evaluated before the upper bound.

Array bound checking for each index of an array is performed for an array specified in the comment directive checkon (see the C option in chapter 6).

The maximum number of dimensions permitted in the declaration of an array is 63.

Own arrays with dynamic bounds (bounds which are not constant in the program) are not permitted. Thus the following declaration, given as an example in section 5.2.2, is illegal:

```
own integer array A [ if C < 0 then 2 else 1 :20]
```

5.2.4 LOWER UPPER BOUND EXPRESSIONS

5.2.4.1 The expressions will be evaluated in the same way as subscript expressions (cf. section 3.1.4.2.).

5.2.4.2 The expressions can only depend on variables and procedures which are non local to the block for which the array declaration is valid. Consequently in the outermost block of a program only array declarations with constant bounds may be declared.

5.2.4.3 An array is defined only when the values of all upper subscript bounds are not smaller than those of the corresponding lower bounds.

5.2.4.4 The expressions will be evaluated once at each entrance into the block.

5.2.5 THE IDENTITY OF SUBSCRIPTED VARIABLES

The identity of a subscripted variable is not related to the subscript bounds given in the array declaration. However, even if an array is declared own the values of the corresponding subscripted variables will, at any time, be defined only for those of these variables which have subscripts within the most recently calculated subscript bounds.

5.3 SWITCH DECLARATIONS

5.3.1 SYNTAX

< switch list > ::= < designational expression > | < switch list > , < designational expression >

< switch declaration > ::= switch < switch identifier > := < switch list >

5.3.2 EXAMPLES

switch S := S1,S2,Q[m] ,if v > -5 then S3 else S4

switch Q:= p1,w

5.3.3 SEMANTICS

A switch declaration defines the set of values of the corresponding switch designators. These values are given one by one as the values of the designational expressions entered in the switch list. With each of these designational expressions there is associated a positive integer, 1,2,..., obtained by counting the items in the list from left to right. The value of the switch designator corresponding to a given value of the subscript expression (cf. section 3.5. Designational Expressions) is the value of the designational expression in the switch list having this given value as its associated integer.

5.3.4 EVALUATION OF EXPRESSIONS IN THE SWITCH LIST

An expression in the switch list will be evaluated every time the item of the list in which the expression occurs is referred to, using the current values of all variables involved.

5.3.5 INFLUENCE OF SCOPES

If a switch designator occurs outside the scope of a quantity entering into a designational expression in the switch list, and an evaluation of this switch designator selects this designational expression, then the conflicts between the identifiers for the quantities in this expression and the identifiers whose declarations are valid at the place of the switch designator will be avoided through suitable systematic changes of the latter identifiers.

5.4 PROCEDURE DECLARATIONS

5.4.1 SYNTAX

< formal parameter > ::= < identifier >

< formal parameter list > ::= < formal parameter > |

< formal parameter list > < parameter delimiter > < formal parameter >

$\langle \text{formal parameter part} \rangle ::= \langle \text{empty} \rangle \mid (\langle \text{formal parameter list} \rangle)$
 $\langle \text{identifier list} \rangle ::= \langle \text{identifier} \rangle \mid \langle \text{identifier list} \rangle , \langle \text{identifier} \rangle$
 $\langle \text{value part} \rangle ::= \underline{\text{value}} \langle \text{identifier list} \rangle ; \mid \langle \text{empty} \rangle$
 $\langle \text{specifier} \rangle ::= \underline{\text{string}} \mid \langle \text{type} \rangle \mid \underline{\text{array}} \mid \langle \text{type} \rangle \underline{\text{array}} \mid \underline{\text{label}} \mid \underline{\text{switch}} \mid \underline{\text{procedure}} \mid \langle \text{type} \rangle \underline{\text{procedure}}$
 $\langle \text{specification part} \rangle ::= \langle \text{empty} \rangle \mid \langle \text{specifier} \rangle \langle \text{identifier list} \rangle ; \mid$
 $\quad \langle \text{specification part} \rangle \langle \text{specifier} \rangle \langle \text{identifier list} \rangle ;$
 $\langle \text{procedure heading} \rangle ::= \langle \text{procedure identifier} \rangle \langle \text{formal parameter part} \rangle ; \langle \text{value part} \rangle \langle \text{specification part} \rangle$
 $\langle \text{procedure body} \rangle ::= \langle \text{statement} \rangle \mid \langle \text{code} \rangle$
 $\langle \text{procedure declaration} \rangle ::= \underline{\text{procedure}} \langle \text{procedure heading} \rangle \langle \text{procedure body} \rangle \mid$
 $\quad \langle \text{type} \rangle \underline{\text{procedure}} \langle \text{procedure heading} \rangle \langle \text{procedure body} \rangle$

5.4.1 Syntax

The following definition of code is added:

$\langle \text{code number} \rangle ::= \langle \text{sequence of 1 to 5 digits} \rangle$
 $\langle \text{external identifier} \rangle ::= \langle \text{identifier containing less than or equal to 7 letters and digits, beginning with a letter} \rangle$
 $\langle \text{code identifier} \rangle ::= \langle \text{code number} \rangle \mid \langle \text{external identifier} \rangle \mid \langle \text{empty} \rangle$
 $\langle \text{code} \rangle ::= \underline{\text{code}} \langle \text{code identifier} \rangle \mid \underline{\text{algo}} \langle \text{code identifier} \rangle \mid$
 $\quad \underline{\text{fortran}} \langle \text{code identifier} \rangle$

For the semantics of code see section 5.4.6.

5.4.2 EXAMPLES (SEE ALSO THE EXAMPLES AT THE END OF THE REPORT)

```

procedure Spur (a) Order: (n) Result: (s); value n;
array a; integer n; real s;
begin integer k;
s:=0;
for k:=1 step 1 until n do s:=s+a [k,k]
end

```

```

procedure Transpose (a) Order: (n) ; value n ;
array a ; integer n ;
begin real w ; integer i,k ;
for i:=1 step 1 until n do
  for k:=1+i step n until n do
    begin w := a [i,k] ;
      a [i,k] := a [k,i]
      a [k,i] := w
    end
  end
end Transpose

```

```

integer procedure Step(u) ; real u ;
Step:=if 0 ≤ u ∧ u ≤ 1 then 1 else 0

```

```

procedure Absmax (a) size: (n,m) Result : (y) Subscripts : (i,k) ;
comment The absolute greatest element of the matrix a, of size n by m is
transferred to y, and the subscripts of this element to i and k ;
array a ; integer n,m,i,k ; real y ;
begin integer p,q ;
y := 0 ;
for p := 1 step 1 until n do for q := 1 step 1 until m do
if abs (a[p,q]) > y then begin y := abs (a[p,q]) ; i := p ;
  k:=q
end
end end Absmax

```

```

procedure Innerproduct (a,b) Order: (k,p) Result: (y) ; value k ;
integer k,p ; real y,a,b ;
begin real s ;
s:=0;
for p:= 1 step 1 until k do s:=s+a×b ;
y:=s
end Innerproduct

```

5.4.3 SEMANTICS

A procedure declaration serves to define the procedure associated with a procedure identifier. The principal constituent of a procedure declaration is a statement or a piece of code, the procedure body, which through the use of procedure statements and/or function designators may be activated from other parts of the block in the head of which the procedure declaration appears. Associated with the body is a heading, which specifies certain identifiers occurring within the body to represent formal parameters. Formal parameters in the procedure

body will, whenever the procedure is activated (cf. section 3.2 Function Designators and section 4.7 Procedure Statements) be assigned the values of or be replaced by actual parameters. Identifiers in the procedure body which are not formal will be either local or nonlocal to the body depending on whether they are declared within the body or not. Those of them which are nonlocal to the body may well be local to the block in the head of which the procedure declaration appears. The procedure body always acts like a block, whether it has the form of one or not. Consequently the scope of any label labelling a statement within the body or the body itself can never extend beyond the procedure body. In addition, if the identifier of a formal parameter is declared anew within the procedure body (including the case of its use as a label as in section 4.1.3.), it is thereby given a local significance and actual parameters which correspond to it are inaccessible throughout the scope of this inner local quantity.

5.4.3 Semantics

The maximum number of formal parameters permitted in the declaration of a procedure is 63. An identifier must not occur more than once in a formal parameter list.

5.4.4 VALUES OF FUNCTION DESIGNATORS

For a procedure declaration to define the value of a function designator, there must, within the procedure body, occur one or more explicit assignment statements with the procedure identifier in the left part; at least one of these must be executed, and the type associated with the procedure identifier must be declared through the appearance of a type declarator as the very first symbol of the procedure declaration. The last value so assigned is used to continue the evaluation of the expression in which the function designator occurs. Any occurrence of the procedure identifier within the body of the procedure other than in a left part in an assignment statement denotes activation of the procedure.

5.4.4 Values of function designators

Since function designators may occur in subscripts, this section should read “. . . as a left part. . .” rather than “. . . in a left part”. This appears twice.

5.4.5 SPECIFICATIONS

In the heading of a specification part, giving information about the kinds and types of the formal parameters by means of an obvious notation, may be included. In this part no formal parameter may occur more than once. Specifications of formal parameters called by value (cf. section 4.7.3.1) must be supplied and specifications of formal parameters called by name (cf. section 4.7.3.2) may be omitted.

5.4.5 Specifications

For ALGOL 4, the last sentence should be changed to read “. . . and specifications of all formal parameters, whether called by name or by value, must be supplied.”

5.4.6 CODE AS PROCEDURE BODY

It is understood that the procedure body may be expressed in non-ALGOL language. Since it is intended that the use of this feature should be entirely a question of hardware representation, no further rules concerning this code language can be given within the reference language.

5.4.6 Code as procedure body

All procedures, unless standard, must be declared in the program in which they are called. In particular, when a program references a separately compiled procedure, the program must contain a declaration of that procedure. This declaration simply consists of a procedure heading and a code procedure body.

The procedure heading has the same format as a normal procedure heading (section 5.4.1) and the code procedure body (defined as code in section 5.4.1) consists of the symbol code, algol, or fortran followed by a code identifier. This may be either a number in the range 0-99999, or name of seven or fewer characters, or empty. If the name has more than 7 letters or digits, only the first 7 are used. If the code identifier is empty, the name of the procedure in which it is declared is assumed to be the name of the external identifier.

The code identifier is either the same number or the same external identifier associated with the procedure when it is compiled separately as an ALGOL or FORTRAN Extended source procedure. The compiler uses this number or external identifier to link the declaration with the procedure. The procedure is linked to the main program when the object program is loaded. For a procedure that utilizes the symbol code or algol, the object code does not have to be produced by the compilation of an ALGOL source procedure; it may be generated in any way provided the handling of formal parameters conforms to the object code produced by the compilation of an ALGOL source procedure. The macros of ALGTEXT may be used for this purpose (see appendix B).

The identifying name included in the procedure heading need not be the same as the name declared in a separately compiled procedure, as linking is done by code number or by external identifier. However, all references in the program must use the name declared in the program rather than the name declared in the separately declared procedure. When the code identifier is empty, the name in the procedure heading is taken to be the external identifier and therefore must agree with the name of the separately declared procedure.

Only simple variables and arrays can be passed as FORTRAN parameters. When passing parameters between ALGOL and FORTRAN, the user should be aware of two major differences in internal representation of data. One is that although ALGOL integers are represented internally in floating point format (section 14.2.1), FORTRAN integers are represented in fixed point format. The other is that elements of ALGOL arrays are stored consecutively so that the rightmost subscript varies most rapidly (section 14.3.2), but FORTRAN array elements are stored with the leftmost subscript varying most rapidly.

To define an external source procedure, the source procedure has the form of a procedure declaration preceded by:

code < code number >; or code < external identifier >;
algol < code number >; or algol < external identifier >;

To call an external source procedure, the calling source procedure or source program must contain a declaration for that procedure of the form:

procedure < procedure heading > < code >
procedure < code procedure heading > < code >
< type > procedure < procedure heading > < code >
< type > procedure < code procedure heading > < code >

where < code procedure heading > ::= < procedure identifier > < formal parameter part >

The declaration can contain specifications of all formal parameters of the procedure; these specifications have no effect.

In Example 1, the procedures AVERAGE and SQUAREAVERAGE are compiled in the original program. In Example 2 they are compiled separately.

Example 1.

```
begin
  real procedure AVERAGE (LOWER, UPPER);
    value LOWER, UPPER;
    real LOWER, UPPER;
    begin AVERAGE:=(LOWER+UPPER)/2
    end;
  real procedure SQUAREAVERAGE (LOW, HIGH);
    value LOW, HIGH;
    real LOW, HIGH;
    begin SQUAREAVERAGE:=SQRT (LOW ↑2+HIGH ↑2)/2
    end;
  real X, Y, S, SQ;
  S := 0;
  SQ := 0;
  for X := 1 step 1 until 100 do
    begin
      Y := X + 1;
      S := S + AVERAGE (X, Y);
      SQ := SQ + SQUAREAVERAGE (X, Y)
    end
  end
```

In the second example the first procedure body is replaced by the symbol code with the identifying number 129. In the heading, the identifying name AVERAGE has been changed to MEAN, and the formal parameter names to A and B. References to this procedure are to the name MEAN. The procedure called AVERAGE should be compiled separately with the code number 129.

The source deck (chapter 4) for the compilation of AVERAGE is:

```
code 129;  
real procedure AVERAGE (LOWER, UPPER);  
  value LOWER, UPPER;  
  real LOWER, UPPER;  
  begin AVERAGE := (LOWER + UPPER)/ 2;  
end;
```

followed by the eop.

The second procedure body is replaced by the symbol code and the identifying number 527. The procedure heading remains the same. Since the identifying name is not changed, the procedure is referenced as before. The procedure called SQUAREAVERAGE should be compiled separately with the code number 527.

Example 2.

```
begin  
  real procedure MEAN (A,B);  
    value A, B;  
    real A, B;  
    code 129;  
  
  real procedure SQUAREAVERAGE (LOW, HIGH);  
    value LOW, HIGH;  
    real LOW, HIGH;  
    code 527;  
  
  real X, Y, S, SQ;  
  S := 0;  
  SQ := 0;  
  for X := 1 step 1 until 100 do  
    begin  
      Y := X + 1;  
      S := S + MEAN (X,Y);  
      SQ := SQ + SQUAREAVERAGE (X,Y)  
    end  
  
end
```

The following example depicts a source procedure with the external identifier FAC:

```
algol FAC ; integer procedure FACTORIAL (n) ;  
    value n ; integer n ;  
    FACTORIAL := if n = 0 then 1  
                else n × FACTORIAL (n-1) ;
```

The external identifier and procedure identifier can be the same.

The external procedure above can be declared by the following declaration:

```
integer procedure ABC (n) ; value n ; integer n ; algol FAC ;
```

All references to the procedure within the source program or source procedure would be to ABC.

The facility offered by the external name makes possible a library of object program procedures compiled with arbitrary identifiers but distinguishing external names.

The names of the formal parameters in the procedure heading need not be the same as those declared when the procedure is compiled separately, but the number of parameters must be the same.

The above rules also apply to a separately compiled procedure which references another separately compiled procedure. The referencing procedure must contain a declaration for the referenced procedure as described above.

The above external procedure could also have been declared by the following declaration:

```
integer procedure FAC (n) ; value n ; integer n ; algol ;
```

Since the code identifier following algol is empty, FAC, the name declared in the procedure heading, is used.

Examples of Procedure Declarations

Example 1.

```
procedure euler (fct, sum, eps, tim) ; value eps, tim ;  
integer tim ; real procedure fct ; real sum, eps;
```

comment euler computes the sum of fct (i) for i from zero up to infinity by means of a suitable refined euler transformation. The summation is stopped as soon as tim times in succession the absolute value of the terms of the transformed series are found to be less than eps. Hence, one should provide a function fct with one integer argument, an upper bound eps, and an integer tim. The output is the sum sum. euler is particularly efficient in the case of slowly convergent or divergent alternating series ;

```

begin integer i,k,n,t; array m [0:15] ; real mn,mp,ds

i := n := t := 0 ; m[0] := fct[0] ; sum := m[0] /2 ;

nextterm:i:=i+1 ; mn := fct(i) ;

    for k := 0 step 1 until n do

        begin mp := (mn + m[k])/2 ; m[k] := mn ;

        mn := mp end means ;

    if (abs(mn) < abs(m[n]))  $\wedge$  (n < 15) then

        begin ds := mn/2 ; n := n+1 ; m[n] :=

        mn end accept

    else ds := mn ;

    sum := sum + ds ;

    if abs(ds) < eps then t := t+1 else t := 0 ;

    if t < tim then go to nextterm

end euler

```

Example 2.[†]

```

procedure RK(x,y,n,FKT,eps,eta,xE,yE,fi) ; value x,y ;

integer n ; Boolean fi ; real x,eps,eta,xE ; array

y,yE ; procedure FKT ;

```

comment: RK integrates the system $Y_k' = f_k(x, y_1, y_2, \dots, y_n)$ ($k = 1, 2, \dots, n$) of differential equations with the method of Runge-Kutta with automatic search for appropriate length of integration step. Parameters are: the initial values x and y [k] for x and the unknown functions $y_k(x)$. The order n of the system. The procedure FKT(x,y,n,z) which represents the system to be integrated, i.e. the set of functions f_k . The tolerance values eps and eta which govern the accuracy of the numerical integration. The end of the integration internal xE. The output parameter yE which represents the solution at $x = xE$. The Boolean variable fi, which must

[†]This RK-program contains some new ideas which are related to ideas of S. Gill, A process for the step-by-step integration of differential equations in an automatic computing machine, [Proc. Camb. Phil. Soc. 47 (1951), 96] ; and E. Froberg, On the solution of ordinary differential equations with digital computing machines, [Fysiograf. Sällsk; Lund, Förh. 20,11 (1950), 136-152]. It must be clear, however, that with respect to computing time and round-off errors it may not be optimal, nor has it actually been tested on a computer.

always be given the value true for an isolated or first entry into RK. If however the function y must be available at several meshpoints x_0, x_1, \dots, x_n , then the procedure must be called repeatedly (with $x=x_k$, $xE=x_{k+1}$, for $k=0, 1, \dots, n-1$) and then the later calls may occur with fi=false which saves computing time. The input parameters of FKT must be x,y,n, the output parameter z represents the set of derivatives $z[k] = f_k(x, y[1], y[2], \dots, y[n])$ for x and the actual y's. A procedure comp enters as a nonlocal identifier;

begin

array z,y1,y2,y3[1:n] ; real x1,x2,x3,H ; Boolean out ;

integer k,j ; own real s,Hs ;

procedure RK1ST (x,y,h,x_e,y_e) ; real x,h,x_e ; array

y,y_e ;

comment: RK1ST integrates one single RUNGE-KUTTA with initial values x,y[k] which yields the output parameters x_e=x+h and y_e[k], the latter being the solution at x_e. Important: the parameters n, FKT, z enter RK1ST as nonlocal entities ;

begin

array w[1:n] , a[1:5] ; integer k,j ;

a[1] := a[2] := a[5] := h/2 ; a[3] := a[4] := h ;

x_e := x ;

for k := 1 step 1 until n do y_e[k] := w[k] := y[k] ;

for j:= 1 step 1 until 4 do

begin

FKT(x_e,w,n,z) ;

x_e := x+a[j] ;

for k := 1 step 1 until n do

begin

w[k] := y[k] + a[j] × z[k] ;

y_e[k] := y_e[k] + a[j+1] × z[k] /3

end k

end j

end RK1ST ;

Begin of program:

if fi then begin $H := xE - x$; $s := 0$ end else $H := Hs$;

$out := \underline{false}$;

AA: if $(x + 2.01 \times H - xE > 0) \equiv (H > 0)$ then

begin $Hs := H$; $out := \underline{true}$; $H := (xE - x)/2$

end if ;

RK1ST $(x, y, 2 \times H, x1, y1)$;

BB: RK1ST $(x, y, H, x2, y2)$; RK1ST $(x2, y2, H, x3, y3)$;

for $k := 1$ step 1 until n do

if $\text{comp}(y1[k], y3[k], \text{eta}) > \text{eps}$ then go to CC ;

comment: $\text{comp}(a, b, c)$ is a function designator, the value of which is the absolute value of the difference of the mantissae of a and b , after the exponents of these quantities have been made equal to the largest of the exponents of the originally given parameters a, b, c ;

$x := x3$; if out then go to DD ;

for $k := 1$ step 1 until n do $y[k] := y3[k]$;

if $s=5$ then begin $s := 0$; $H := 2 \times H$ end if ;

$s := s + 1$; go to AA ;

CC: $H := 0.5 \times H$; $out := \underline{false}$; $x1 := x2$

for $k := 1$ step 1 until n do $y1[k] := y2[k]$;

go to BB ;

DD: for $k := 1$ step 1 until n do $yE[k] := y3[k]$;

end RK

ALPHABETIC INDEX OF DEFINITIONS OF CONCEPTS AND SYNTACTIC UNITS

All references are given through section numbers. The references are given in three groups:

- def Following the abbreviation "def", reference to the syntactic definition (if any) is given.
- synt Following the abbreviation "synt", references to the occurrences in metalinguistic formulae are given. References already quoted in the def-group are not repeated.
- text Following the word "text", the references to definitions given in the text are given.

The basic symbols represented by signs other than underlined words have been collected at the beginning.

The examples have been ignored in compiling the index.

- $+$, see: plus
- $-$, see: minus
- \times , see: multiply
- $/, \div$, see: divide
- \uparrow , see: exponentiation
- $<, \leq, =, \geq, >, \neq$, see: <relational operator>
- $\equiv, \supset, \vee, \wedge, \neg$, see: <logical operator>
- $,$, see: comma
- $.$, see: decimal point
- 10 , see: ten
- $:$, see: colon
- $;$, see: semicolon
- $:=$, see: colon equal
- \sqcup , see: space
- $()$, see: parentheses
- $[]$, see: subscript brackets
- $'$, see: string quotes
- < actual parameter >, def 3.2.1, 4.7.1
- < actual parameter list >, def 3.2.1, 4.7.1
- < actual parameter part >, def 3.2.1, 4.7.1
- < adding operator >, def 3.3.1
 - alphabet, text 2.1
 - arithmetic, text 3.3.6
- < arithmetic expression >, def 3.3.1 synt 3, 3.1.1, 3.3.1, 3.4.1, 4.2.1, 4.6.1, 5.2.1 text 3.3.3
- < arithmetic operator >, def 2.3 text 3.3.4
 - array, synt 2.3, 5.2.1, 5.4.1
 - array, text 3.1.4.1
- < array declaration >, def 5.2.1 synt 5 text 5.2.3
- < array identifier >, def 3.1.1 synt 3.2.1, 4.7.1, 5.2.1 text 2.8
- < array list >, def 5.2.1
- < array segment >, def 5.2.1
- < assignment statement >, def 4.2.1 synt 4.1.1 text 1, 4.2.3
- < basic statement >, def 4.1.1 synt 4.5.1
- < basic symbol >, def 2
 - begin, synt 2.3, 4.1.1
- < block >, def 4.1.1 synt 4.5.1 text 1, 4.1.3, 5
- < block head >, def 4.1.1
 - Boolean, synt 2.3, 5.1.1 text 5.1.3
- < Boolean expression >, def 3.4.1 synt 3, 3.3.1, 4.2.1, 4.5.1, 4.6.1 text 3.4.3
- < Boolean factor >, def 3.4.1
- < Boolean primary >, def 3.4.1
- < Boolean secondary >, def 3.4.1
- < Boolean term >, def 3.4.1
- < bound pair >, def 5.2.1
- < bound pair list >, def 5.2.1
- < bracket >, def 2.3
- < code >, synt 5.4.1 text 4.7.8, 5.4.6
 - colon :, synt 2.3, 3.2.1, 4.1.1, 4.5.1, 4.6.1, 4.7.1, 5.2.1
 - colon equal :=, synt 2.3, 4.2.1, 4.6.1, 5.3.1
 - comma ,, synt 2.3, 3.1.1, 3.2.1, 4.6.1, 4.7.1, 5.1.1, 5.2.1, 5.3.1, 5.4.1
 - comment, synt 2.3
 - comment convention, text 2.3
- < compound statement >, def 4.1.1 synt 4.5.1 text 1
- < compound tail >, def 4.1.1
- < conditional statement >, def 4.5.1 synt 4.1.1 text 4.5.3
- < decimal fraction >, def 2.5.1
- < decimal number >, def 2.5.1 text 2.5.3
 - decimal point ., synt 2.3, 2.5.1,
- < declaration >, def 5 synt 4.1.1 text 1, 5 (complete section)
- < declarator >, def 2.3
- < delimiter >, def 2.3 synt 2
- < designational expression >, def 3.5.1 synt 3, 4.3.1, 5.3.1 text 3.5.3

< digit > , def 2.2.1 synt 2, 2.4.1, 2.5.1
 dimension, text 5.2.3.2
 divide / ÷, synt 2.3, 3.3.1 text 3.3.4.2
 do, synt 2.3, 4.6.1
 < dummy statement > , def 4.4.1 synt 4.1.1 text 4.4.3
 else, synt 2.3, 3.3.1, 3.4.1, 3.5.1, 4.5.1 text 4.5.3.2
 < empty > , def 1.1 synt 2.6.1, 3.2.1, 4.4.1, 4.7.1, 5.4.1
 end, synt 2.3, 4.1.1
 entier, text 3.2.5
 exponentiation ↑, synt 2.3, 3.3.1 text 3.3.4.3
 < exponent part > , def 2.5.1 text 2.5.3
 < expression > , def 3 synt 3.2.1, 4.7.1 text 3
 (complete section)

 < factor > , def 3.3.1
 false, synt 2.2.2
 for, synt 2.3, 4.6.1
 < for clause > , def 4.6.1 text 4.6.3
 < for list > , def 4.6.1 text 4.6.4
 < for list element > , def 4.6.1 text 4.6.4.1, 4.6.4.2,
 4.6.4.3
 < formal parameter > , def 5.4.1 text 5.4.3
 < formal parameter list > , def 5.4.1
 < formal parameter part > , def 5.4.1
 < for statement > , def 4.6.1 synt 4.1.1, 4.5.1 text 4.6
 (complete section)
 < function designator > , def 3.2.1 synt 3.3.1, 3.4.1
 text 3.2.3, 5.4.4

 go to, synt 2.3, 4.3.1
 < go to statement > , def 4.3.1 synt 4.1.1 text 4.3.3

 < identifier > , def 2.4.1 synt 3.1.1, 3.2.1, 3.5.1, 5.4.1
 text 2.4.3
 < identifier list > , def 5.4.1
 if, synt 2.3, 3.3.1, 4.5.1
 < if clause > , def 3.3.1, 4.5.1 synt 3.4.1, 3.5.1
 text 3.3.3, 4.5.3.2
 < if statement > , def 4.5.1 text 4.5.3.1
 < implication > , def 3.4.1
 integer, synt 2.3, 5.1.1 text 5.1.3
 < integer > , def 2.5.1 text 2.5.4

 label, synt 2.3, 5.4.1
 < label > , def 3.5.1 synt 4.1.1, 4.5.1, 4.6.1 text 1,
 4.1.3
 < left part > , def 4.2.1
 < left part list > , def 4.2.1

 < letter > , def 2.1 synt 2, 2.4.1, 3.2.1, 4.7.1
 < letter string > , def 3.2.1, 4.7.1
 local, text 4.1.3
 < local or own type > , def 5.1.1 synt 5.2.1
 < logical operator > , def 2.3 synt 3.4.1 text 3.4.5
 < logical value > , def 2.2.2 synt 2, 3.4.1
 < lower bound > , def 5.2.1 text 5.2.4

 minus −, synt 2.3, 2.5.1, 3.3.1 text 3.3.4.1
 multiply ×, synt 2.3, 3.3.1 text 3.3.4.1
 < multiplying operator > , def 3.3.1

 nonlocal, text 4.1.3
 < number > , def 2.5.1 text 2.5.3, 2.5.4

 < open string > , def 2.6.1
 < operator > , def 2.3
 own, synt 2.3, 5.1.1 text 5, 5.2.5

 < parameter delimiter > , def 3.2.1, 4.7.1 synt 5.4.1
 text 4.7.7
 parentheses (), synt 2.3, 3.2.1, 3.3.1, 3.4.1, 3.5.1,
 4.7.1, 5.4.1 text 3.3.5.2
 plus +, synt 2.3, 2.5.1, 3.3.1 text 3.3.4.1
 < primary > , def 3.3.1
 procedure, synt 2.3, 5.4.1
 < procedure body > , def 5.4.1
 < procedure declaration > , def 5.4.1 synt 5 text 5.4.3
 < procedure heading > , def 5.4.1 text 5.4.3
 < procedure identifier > , def 3.2.1 synt 3.2.1, 4.7.1,
 5.4.1 text 4.7.5.4
 < procedure statement > , def 4.7.1 synt 4.1.1 text 4.7.3
 < program > , def 4.1.1 text 1
 < proper string > , def 2.6.1

 quantity, text 2.7

 real, synt 2.3, 5.1.1 text 5.1.3
 < relation > , def 3.4.1 text 3.4.5
 < relational operator > , def 2.3, 3.4.1

 scope, text 2.7
 semicolon ; , synt 2.3, 4.1.1, 5.4.1
 < separator > , def 2.3
 < sequential operator > , def 2.3
 < simple arithmetic expression > , def 3.3.1 text 3.3.3
 < simple Boolean > , def 3.4.1
 < simple designational expression > , def 3.5.1
 < simple variable > , def 3.1.1 synt 5.1.1 text 2.4.3
 space ␣, synt 2.3 text 2.3, 2.6.3

< specification part > , def 5.4.1 text 5.4.5
 < specifier > , def 2.3
 < specifier > , def 5.4.1
 standard function, text 3.2.4, 3.2.5
 < statement > , def 4.1.1 synt 4.5.1, 4.6.1, 5.4.1 text 4
 (complete section)
 statement bracket, see: begin end
 step, synt 2.3, 4.6.1 text 4.6.4.2
 string, synt 2.3, 5.4.1
 < string > , def 2.6.1 synt 3.2.1, 4.7.1 text 2.6.3
 string quotes^(*), synt 2.3, 2.6.1 text 2.6.3
 subscript, text 3.1.4.1
 subscript bound, text 5.2.3.1
 subscript brackets [], synt 2.3, 3.1.1, 3.5.1, 5.2.1
 < subscripted variable > , def 3.1.1 text 3.1.4.1
 < subscript expression > , def 3.1.1 synt 3.5.1
 < subscript list > , def 3.1.1
 successor, text 4
 switch, synt 2.3, 5.3.1, 5.4.1
 < switch declaration > , def 5.3.1 synt 5 text 5.3.3
 < switch designator > , def 3.5.1 text 3.5.3
 < switch identifier > , def 3.5.1 synt 3.2.1, 4.7.1, 5.3.1
 < switch list > , def 5.3.1

 < term > , def 3.3.1
 ten₁₀ , synt 2.3, 2.5.1

 then, synt 2.3, 3.3.1, 4.5.1
 transfer function, text 3.2.5
 true, synt 2.2.2
 < type > , def 5.1.1 synt 5.4.1 text 2.8
 < type declaration > , def 5.1.1 synt 5 text 5.1.3
 < type list > , def 5.1.1

 < unconditional statement > , def 4.1.1, 4.5.1
 < unlabelled basic statement > , def 4.1.1
 < unlabelled block > , def 4.1.1
 < unlabelled compound > , def 4.1.1
 < unsigned integer > , def 2.5.1, 3.5.1
 < unsigned number > , def 2.5.1 synt 3.3.1
 until, synt 2.3, 4.6.1 text 4.6.4.2
 < upper bound > , def 5.2.1 text 5.2.4

 value, synt 2.3, 5.4.1
 value, text 2.8, 3.3.3
 < value part > , def 5.4.1, text 4.7.3.1
 < variable > , def 3.1.1 synt 3.3.1, 3.4.1, 4.2.1, 4.6.1
 text 3.1.3
 < variable identifier > , def 3.1.1

 while, synt 2.3, 4.6.1 text 4.6.4.3

The index is expanded as follows:

algol, synt 2.3, 4.7.8, 5.4.6
fortran synt 2.3, 4.7.8, 5.4.6
 external identifier, def 2.4.1, 5.4.1 text 5.4.6
virtual, text 2.3.1
snapoff, text 2.3.1
snap, text 2.3.1
trace, text 2.3.1
overlay, text 2.3.1

The processes of input and output deal with the mapping of basic characters onto input and output devices under the control of format rules. Characters are grouped to form lines, and lines are grouped to form pages. A page consists of printed lines and a line may be a printed line or card image.

The relation between lines and pages and physical entities (such as records and blocks) depends on formatting rules, channel specifications (B.1.1), input/output procedures, and the physical device involved in the input/output process. The user need not, in general, be aware of the details of this relationship, since the input/output process is symmetric. Given the same specifications, a file output by the system is disassembled into the same lines and pages as input by the system.

3.1 COMPARISON WITH ACM PROPOSAL FOR INPUT/OUTPUT

The following descriptions explain the differences between the input/output procedures included in ALGOL 4 and the procedures defined in the ACM proposal.[†] To facilitate cross referencing, the same numbering system is used in this chapter as in the proposal. The ACM proposal is a continuation of the ALGOL-60 Revised Report, and should be considered a continuation of Chapter 2 of this manual.

All descriptions of the modifications to the input/output procedures are made at the main reference in the proposal; and wherever feasible, all other references are noted. The reader should assume, however, that such modifications apply to all references to the features, noted or otherwise.

A section or feature not mentioned in this chapter is implemented, in ALGOL 4, in exact accordance with the proposal.

This chapter also contains descriptions of additional input/output procedures which are not defined in the ACM proposal, and a description of the transmission error and end of partition capabilities automatically supplied within the framework of the input/output procedures.

[†]“A Proposal for Input-Output Conventions in ALGOL-60”, published in The Communications of the ACM, vol. 7 no. 5, May 1964.

A Proposal for Input-Output Conventions in ALGOL-60

A Report of the Subcommittee on ALGOL of the ACM Programming Languages Committee

D. E. Knuth, Chairman

L. L. Bumgarner P. Z. Ingerman J. N. Merner

D. E. Hamilton M. P. Lietzke D. T. Ross

The ALGOL-60 language as first defined made no explicit reference to input and output processes. Such processes appeared to be quite dependent on the computer used, and so it was difficult to obtain agreement on those matters. As time has passed, a great many ALGOL compilers have come into use, and each compiler has incorporated some input-output facilities. Experience has shown that such facilities can be introduced in a manner which is compatible and consistent with the ALGOL language, and which (more importantly) is almost completely machine-independent. However, the existing implementations have taken many different approaches to the subject, and this has hampered the interchange of programs between installations. The ACM ALGOL committee has carefully studied the various proposals in an attempt to define a set of conventions for doing input and output which would be suitable for use on most computers. The present report constitutes the recommendations of that committee.

The input-output conventions described here do not involve extensions or changes to the ALGOL-60 language. Hence they can be incorporated into existing processors with a minimum of effort. The conventions take the form of a set of procedures,¹ which are to be written in code for the various machines; this report discusses the function and use of these procedures. The material contained in this proposal is intended to supplement procedures in real, out real, in symbol, out symbol which have been defined by the international ALGOL committee; the procedures described here could, with trivial exceptions, be expressed in terms of these four.²

The first part of this report describes the methods by which formats are represented; then the calls on the input and output procedures themselves are discussed. The primary objective of the present report is to describe the proposal concisely and precisely, rather than to give a programmer's introduction to the input-output conventions. A simpler and more intuitive (but less exact) description can be written to serve as a teaching tool.

Many useful ideas were suggested by input-output conventions of the compilers listed in the references below. We are also grateful for the extremely helpful contributions of F. L. Bauer, M. Paul, H. Rutishauser, K. Samelson, G. Seegmüller, W. L. v.d. Poel, and other members of the European computing community, as well as A. Evans, Jr., R. W. Floyd, A. G. Grace, J. Green, G. E. Haynam, and W. C. Lynch of the USA.

A. Formats

In this section a certain type of string, which specifies the format of quantities to be input or output, is defined, and its meaning is explained.

A. Formats

A format string can be a string or an array into which a string has been read, using H format (section A.2.3.3). Whenever a format string can be specified, either form can be used. When a format string is contained in an array allocated in LCM, the array is called by value by any library input/output procedure, otherwise by name.

¹Throughout this report, names of system procedures are in lower case, and names of procedures used in illustrative examples are in upper case.

(NOTE: Additional input-output procedures provided by CONTROL DATA are in upper case.)

²Defined at meeting IFIP/WG2.1 – ALGOL in Delft during September, 1963.

A.1 Number Formats (cf. ALGOL Report 2.5)

A.1.1 Syntax

Basic components:

$\langle \text{replicator} \rangle ::= \langle \text{unsigned integer} \rangle \mid X$

$\langle \text{insertion} \rangle ::= B \mid \langle \text{replicator} \rangle B \mid \langle \text{string} \rangle$

$\langle \text{insertion sequence} \rangle ::= \langle \text{empty} \rangle \mid \langle \text{insertion sequence} \rangle \langle \text{insertion} \rangle$

$\langle Z \rangle ::= Z \mid \langle \text{replicator} \rangle Z \mid Z \langle \text{insertion sequence} \rangle C \mid \langle \text{replicator} \rangle Z \langle \text{insertion sequence} \rangle C$

$\langle Z \text{ part} \rangle ::= \langle Z \rangle \mid \langle Z \text{ part} \rangle \langle Z \rangle \mid \langle Z \text{ part} \rangle \langle \text{insertion} \rangle$

$\langle D \rangle ::= D \mid \langle \text{replicator} \rangle D \mid D \langle \text{insertion sequence} \rangle C \mid$

$\langle \text{replicator} \rangle D \langle \text{insertion sequence} \rangle C$

$\langle D \text{ part} \rangle ::= \langle D \rangle \mid \langle D \text{ part} \rangle \langle D \rangle \mid \langle D \text{ part} \rangle \langle \text{insertion} \rangle$

$\langle T \text{ part} \rangle ::= \langle \text{empty} \rangle \mid T \langle \text{insertion sequence} \rangle$

$\langle \text{sign part} \rangle ::= \langle \text{empty} \rangle \mid \langle \text{insertion sequence} \rangle + \mid$

$\langle \text{insertion sequence} \rangle -$

$\langle \text{integer part} \rangle ::= \langle Z \text{ part} \rangle \mid \langle D \text{ part} \rangle \mid \langle Z \text{ part} \rangle \langle D \text{ part} \rangle$

Format Structures:

$\langle \text{unsigned integer format} \rangle ::= \langle \text{insertion sequence} \rangle \langle \text{integer part} \rangle$

$\langle \text{decimal fraction format} \rangle ::= . \langle \text{insertion sequence} \rangle \langle D \text{ part} \rangle \langle T \text{ part} \rangle \mid$

$V \langle \text{insertion sequence} \rangle \langle D \text{ part} \rangle \langle T \text{ part} \rangle$

$\langle \text{exponent part format} \rangle ::= 10 \langle \text{sign part} \rangle \langle \text{unsigned integer format} \rangle$

$\langle \text{decimal number format} \rangle ::= \langle \text{unsigned integer format} \rangle \langle T \text{ part} \rangle \mid$

$\langle \text{insertion sequence} \rangle \langle \text{decimal fraction format} \rangle \mid$

$\langle \text{unsigned integer format} \rangle \langle \text{decimal fraction format} \rangle$

$\langle \text{number format} \rangle ::= \langle \text{sign part} \rangle \langle \text{decimal number format} \rangle |$
 $\langle \text{decimal number format} \rangle + \langle \text{insertion sequence} \rangle |$
 $\langle \text{decimal number format} \rangle - \langle \text{insertion sequence} \rangle |$
 $\langle \text{sign part} \rangle \langle \text{decimal number format} \rangle \langle \text{exponent part format} \rangle$

Note. This syntax could have been described more simply, but the rather awkward constructions here have been formulated so that no syntactic ambiguities (in the sense of formal language theory) will exist.

A.1.1 Syntax

The letter C is not implemented. All references to C in the ACM Report should be disregarded. For example: $\langle Z \rangle ::= Z | \langle \text{replicator} \rangle Z | Z \langle \text{insertion sequence} \rangle C | \langle \text{replicator} \rangle Z \langle \text{insertion sequence} \rangle C$ is implemented as follows:

$\langle Z \rangle ::= Z | \langle \text{replicator} \rangle Z$

A.1.2 Examples. Examples of number formats appear in Figure 1.

Number format	Result from -13.296	Result from 1007.999
+ZZZCDDD.DD	-013.30	+1,008.00
+3ZC3D.2D	-013.30	+1,008.00
-3D2B3D.2DT	-000 013.29	001 007.99
5Z.5D-	13.29600-	1007.99900
'integer'part' -4ZV	integer part -13,	integer part 1007,
'fraction'B3D	fraction 296	fraction 999
-.5D ₁₀ +2D'...''	-.13296 ₁₀ +02. . .	.10080 ₁₀ +04. . .
+ZD ₁₀ 2Z	-13	+10 ₁₀ 2
+D.DDBDDDBDD ₁₀	-1.32 96 00 10+01	+1.00 79 99 10 +03
+DD		
XB.XD ₁₀ -DDD	(depends on call)	(depends on call)

Figure 1

A.1.2 Examples. Figure 1 (depends on call) – for definition of call see A.1.3.3 Sign and Zero Suppression.

A.1.3 Semantics. The above syntax defines the allowable strings which can comprise a "number format." We will first describe the interpretation to be taken during output.

A.1.3.1 Replicators. An unsigned integer n used as replicator means the quantity is repeated n times; thus 3B is equivalent to BBB. The character X as replicator means a number of times which will be specified when the format is called (see Section B.3.1).

A.1.3.1 Replicators. A replicator of value 0 (n or X) implies the absence of the quantity to which the replicator refers. The maximum size of a replicator is 262,143.

A.1.3.2 Insertions. The syntax has been set up so that strings, delimited by string quotes, may be inserted anywhere within a number format. The corresponding information in the strings (except for the outermost string quotes) will appear inserted in the same place with respect to the rest of the number. Similarly, the letter B may be inserted anywhere within a number format, and it stands for a blank space.

A.1.3.2 Insertions. Insertions can include any of the basic characters allowed in strings (chapter 2, section 2.6.1).

A.1.3.3 Sign, zero, and comma suppression. The portion of a number to the left of the decimal point consists of an optional sign, then a sequence of Z's and a sequence of D's with possible C's following a Z or a D, plus possible insertion characters.

The convention on signs is the following: (a) if no sign appears, the number is assumed to be positive, and the treatment of negative numbers is undefined; (b) if a plus sign appears, the sign will appear as + or – on the external medium; and (c) if a minus sign appears, the sign will appear if minus, and will be suppressed if plus.

The letter Z stands for zero suppression, and the letter D stands for digit printing without zero suppression. Each Z and D stands for a single digit position; a zero digit specified by Z will be suppressed, i.e., replaced by a blank space, when all digits to its left are zero. A digit specified by D will always be printed. Note that the number zero printed with all Z's in the format will give rise to all blank spaces, so at least one D should usually be given somewhere in the format. The letter C stands for a comma. A comma following a D will always be printed; a comma following a Z will be printed except when zero suppression takes place at that Z. Whenever zero or comma suppression takes place, the sign (if any) is printed in place of the rightmost character suppressed.

A.1.3.3 Sign and Zero Suppression. On input, if no sign appears in the format and the number is negative, an error condition exists. If the procedure BAD DATA (section 3.3) has not been called, or if the label established by it is no longer accessible, a fatal error message is issued and the object program terminates abnormally (chapter 8). Otherwise, control is transferred to the BAD DATA label. Output uses the standard format bounded on either side by an asterisk (section A.5). Comma suppression is not implemented.

A.1.3.4 Decimal points. The position of the decimal point is indicated either by the character "." or by the letter V. In the former case, the decimal point appears on the external medium; in the latter case the decimal point is "implied" i.e., it takes up no space on the external medium. (This feature is most commonly used to save time and space when preparing input data). Only D's (no Z's) may appear to the right of the decimal point.

A.1.3.5 Truncation. On output, nonintegral numbers are usually rounded to fit the format specified. If the letter T is used, however, truncation takes place instead. Rounding and truncation of a number X to d decimal places are defined as follows:

Rounding $10^{-d} \text{ entier } (10^d X + 0.5)$

Truncation $10^{-d} \text{ sign } (X) \text{ entier } (10^d \text{ abs } (X))$

A.1.3.5 Truncation. On output, the number of significant digits appearing for a real number cannot exceed the degree of accuracy available in 60-bit floating point form. Thus at most 14 or 15 significant digits are output, followed by trailing zeros, if necessary (section 5.1.3). The letter T has no meaning when applied to an integer number and is ignored.

A.1.3.6 Exponent part. The number following a "10" is treated exactly the same as the portion of a number to the left of a decimal point (Section A.1.3.3) except if the "D" part of the exponent is empty, i.e., no D's appear, and if the exponent is zero, the "10" and the sign are deleted.

A.1.3.6 Exponent part. The number following a "10" is treated exactly the same as the portion of a number to the left of a decimal point (section A.1.3.3) even if the "D" part of the exponent is empty. If no D's appear, and the exponent is zero, the "10" and the sign are not deleted.

A.1.3.7 Two types of numeric format. Number formats are of two principal kinds: (a) Decimal number with no exponent. In this case, the number is aligned according to the decimal point with the picture in the format, and it is then truncated or rounded to the appropriate number of decimal places. The sign may precede or follow the number.

(b). Decimal number with exponent. In this case, the number is transformed into the format of the decimal number with its most significant digit non-zero; the exponent is adjusted accordingly. If the number is zero, both the decimal part and the exponent part are output as zero. If in case (a) the number is too large to be output in the specified form, or if in case (b) the exponent is too large, an overflow error occurs. The action which takes place on overflow is undefined; it is recommended that the number of characters used in the output be the same as if no overflow has occurred, and that as much significant information as possible be output.

A.1.3.7 Two Types of Numeric Format. A maximum of 24 D's and Z's may appear before the exponent part in a number format; in the exponent part, the maximum is 4. On output overflow, the standard format bounded on either side by an asterisk is used (section A.5).

A.1.3.8 Input. A number input with a particular format specification should in general be the same as the number which would be output with the same format, except less error checking occurs. The rules are, more precisely:

(a) leading zeros and commas may appear even though Z's are used in the format. Leading spaces may appear even if D's are used. In other words, no distinction between Z and D is made on input.

(b) insertions take the same amount of space in the same positions, but the characters appearing there are ignored on input. In other words, an insertion specifies only the number of characters to ignore, when it appears in an input format.

(c) If the format specifies a sign at the left, the sign may appear in any Z, D or C position as long as it is to the left of the number. A sign specified at the right must appear in place.

(d) The following things are checked: The position of commas, decimal points, "10" and the presence of digits in place of D or Z after the first significant digit. If an error is detected in the data, the result is undefined; it is recommended that the input procedure attempt to reread the data as if it were in standard format (Section A.5) and also to give some error indication compatible with the system being used. Such an error indication might be suppressed at the programmer's option if the data became meaningful when it was reread in standard format.

A.1.3.8 Input. If an error is found in the data, no attempt is made to reread the data as if it were in standard format; a fatal error diagnostic is issued instead.

If a number is read whose absolute value is larger than the largest number acceptable and its destination is real, then the destination is set to the largest number (cf. chapter 2, section 5.1.3) with the appropriate sign. If a number is read whose absolute value is smaller than the smallest real number distinguishable from zero and its destination is real, then the destination is set to 0.0. If an attempt is made to read a number (real or integer) into a variable of type integer that lies outside the allowable numerical range for type integer data, an error occurs.

The values of the largest number acceptable and the smallest real number distinguishable from zero are available via the standard procedure THRESHOLD (cf. chapter 3, section 3.5).

If the input data does not conform to the format, an error condition exists. If the procedure BAD DATA was not called or if the established label is no longer accessible, a fatal error message is issued and the object program terminates abnormally. Otherwise, control is transferred to the BAD DATA label.

A.2 Other formats

A.2.1 Syntax

$\langle S \rangle ::= S | \langle \text{replicator} \rangle S$

$\langle \text{string format} \rangle ::= \langle \text{insertion sequence} \rangle \langle S \rangle | \langle \text{string format} \rangle \langle S \rangle | \langle \text{string format} \rangle \langle \text{insertion} \rangle$

$\langle A \rangle ::= A | \langle \text{replicator} \rangle A$

$\langle \text{alpha format} \rangle ::= \langle \text{insertion sequence} \rangle \langle A \rangle | \langle \text{alpha format} \rangle \langle A \rangle |$

$\langle \text{alpha format} \rangle \langle \text{insertion} \rangle$

$\langle \text{nonformat} \rangle ::= I | R | L$

$\langle \text{Boolean part} \rangle ::= P | 5F | FFFFF | F$

$\langle \text{Boolean format} \rangle ::= \langle \text{insertion sequence} \rangle \langle \text{Boolean part} \rangle \langle \text{insertion sequence} \rangle$

$\langle \text{title format} \rangle ::= \langle \text{insertion} \rangle | \langle \text{title format} \rangle \langle \text{insertion} \rangle$

$\langle \text{alignment mark} \rangle ::= / | \uparrow | \langle \text{replicator} \rangle / | \langle \text{replicator} \rangle \uparrow$

$\langle \text{format item 1} \rangle ::= \langle \text{number format} \rangle | \langle \text{string format} \rangle |$

$\langle \text{alpha format} \rangle | \langle \text{nonformat} \rangle | \langle \text{Boolean format} \rangle | \langle \text{title format} \rangle |$

$\langle \text{alignment mark} \rangle \langle \text{format item 1} \rangle$

$\langle \text{format item} \rangle ::= \langle \text{format item 1} \rangle | \langle \text{alignment mark} \rangle | \langle \text{format item} \rangle \langle \text{alignment mark} \rangle$

A.2.1 Syntax. The following definitions replace the definitions in the ACM Report:

$\langle S \rangle ::= S | \langle \text{replicator} \rangle S$

$\langle \text{string format} \rangle ::= \langle \text{insertion sequence} \rangle \langle S \rangle | \langle \text{string format} \rangle \langle S \rangle | \langle \text{string format} \rangle \langle \text{insertion} \rangle$

$\langle A \rangle ::= A | \langle \text{replicator} \rangle A$

$\langle \text{alpha format} \rangle ::= \langle \text{insertion sequence} \rangle \langle A \rangle | \langle \text{alpha format} \rangle \langle A \rangle | \langle \text{alpha format} \rangle \langle \text{insertion} \rangle$

$\langle \text{standard format} \rangle ::= N$

$\langle \text{nonformat} \rangle ::= I | R | L | M | H$

$\langle \text{Boolean part} \rangle ::= P | F$

$\langle \text{Boolean format} \rangle ::= \langle \text{insertion sequence} \rangle \langle \text{Boolean part} \rangle \langle \text{insertion sequence} \rangle$

$\langle \text{title format} \rangle ::= \langle \text{insertion} \rangle | \langle \text{title format} \rangle \langle \text{insertion} \rangle$

$\langle \text{alignment mark} \rangle ::= / | \uparrow | J | \langle \text{replicator} \rangle / | \langle \text{replicator} \rangle \uparrow | \langle \text{replicator} \rangle J$

$\langle \text{format item 1} \rangle ::= \langle \text{number format} \rangle | \langle \text{string format} \rangle | \langle \text{alpha format} \rangle | \langle \text{nonformat} \rangle | \langle \text{Boolean format} \rangle |$

$\langle \text{title format} \rangle | \langle \text{alignment mark} \rangle \langle \text{format item 1} \rangle | \langle \text{standard format} \rangle$

$\langle \text{format item} \rangle ::= \langle \text{format item 1} \rangle |$

$\langle \text{alignment mark} \rangle | \langle \text{format item} \rangle \langle \text{alignment mark} \rangle$

A standard format item, N, has been added.

The characters M and H have been added to the non-format codes.

J has been added to alignment mark (section B.3).

A.2.2 Examples

```

↑5Z.5D///
3S'='6S4B
AA'='
↑R
P
/*Execution.*↑

```

A.2.3 Semantics

The maximum length of a format item, after expanding each quantity in it by the corresponding replicator, is 136 characters; the expanded format item corresponds to the data on the external device.

A.2.3.1 String format. A string format is used for output of string quantities. Each of the S-positions in the format corresponds to a single character in the string which is output. If the string is longer than the number of S's, the leftmost characters are transferred; if the string is shorter, ␣-symbols are effectively added at the right of the string.

The word "character" as used in this report refers to one unit of information on the external input or output medium; if ALGOL basic symbols are used in strings which do not have a single-character representation on the external medium being used, the result is undefined.

A.2.3.1 String format. Because of the difference in the definition of a string (section 2.6.1), each of the S-positions in the format corresponds to a single basic character in the output string rather than a single basic symbol. If the length of the string exceeds the number of S's, the leftmost basic characters are transferred; if the string is shorter, blank characters are added to the right.

A.2.3.2 Alpha format. Each letter A means one character is to be transmitted; this is the same as S-format except the ALGOL equivalent of the alphabets is of type integer rather than a string. The translation between the external and internal code will vary from one machine to another, and so programmers should refrain from using this feature in a machine dependent manner. Each implementor should specify the maximum number of characters which can be used for a single integer variable. The following operations are undefined for quantities which have been input using alpha format; arithmetic operations, relations except "=" and "≠", and output using a different number of A's in the output format. If the integer is output using the same number of A's, the same string will be output as was input.

A programmer may work with these alphabetic quantities in a machine-independent manner by using the transfer function equiv(S) where S is a string; the value of equiv(S) is of type integer, and it is defined to have exactly the same value as if the string S had been input using alpha format. For example, one may write

if X = equiv('ALPHA') then go to PROCESS ALPHA

where the value of X has been input using the format "AAAAA".

A.2.3.2 Alpha Format. The A format is the same as S format except that the ALGOL 4 equivalent of the basic character is of type integer rather than string.

In ALGOL 4, the transfer function EQUIV(S) is an integer procedure, the value of which is the internal representation of the first 8 characters in the string S. If the string S contains less than 8 characters, then binary zeroes are added to the right. Thus the value of EQUIV(S) is the same as if the string character had been input under A format.

A.2.3.3 Nonformat. An I, R or L is used to indicate that the value of a single variable of integer, real or Boolean type, respectively, is to be input or output from or to an external medium, using the internal machine representation. If a value of type integer is output with R-format or if a value of type real is input with I-format, the appropriate transfer function is invoked. The precise behaviour of this format, and particularly its interaction with other formats, is undefined in general.

A.2.3.3 Nonformat. The M code added to the nonformat codes indicates that the value of a single variable of any type is to be input or output in the exact form in which it appears on the external device or in memory.

The nonformat codes I, R, L and M each input or output 20 consecutive (6-bit) display octal characters, and map them to or from the 20 consecutive octal (3-bit) digits which constitute one variable internally (Section 5.1.3).

The H code added to the nonformat codes indicates that 8 consecutive display code characters are to be input or output to or from a single integer variable. This can be used to input format strings (cf. section B.3.1).

A.2.3.4 Boolean format. When Boolean quantities are input or output, the format P, F, 5F or FFFFF must be used. The correspondence is defined as follows:

Internal to ALGOL	P	F	5F=FFFFF
<u>true</u>	1	T	TRUE □
<u>false</u>	0	F	FALSE

On input, anything failing to be in the proper form is undefined.

A.2.3.4 Boolean format. When Boolean quantities are input or output, the format P or F must be used. The correspondence is defined as follows: (section A.2.1)

Internal to ALGOL	P	F
<u>true</u>	1	T
<u>false</u>	0	F

External representations in F format are T and F rather than TRUE or FALSE. On input, incorrect forms cause error conditions: if the procedure BADDATA was not called or if established labels are no longer accessible, a fatal error message is issued and the object program terminates abnormally.

A.3.3 Semantics. A format string is simply a list of format items, which are to be interpreted from left to right. The construction "**< replicator > (< format secondary >)**" is simply an abbreviation for "replicator" repetitions of the parenthesized quantity (see Section A.1.3.1). The construction "**(< format secondary >)**" is used to specify an infinite repetition of the parenthesized quantity.

All spaces within a format string except those which are part of insertion substrings are irrelevant.

It is recommended that the ALGOL compiler check the syntax of strings which (from their context) are known to be format strings as the program is compiled. In most cases it will also be possible for the compiler to translate format strings into an intermediate code designed for highly efficient input-output processing by the other procedures.

A.3.3 Semantics. The infinite repetition of the parenthesized quantity is defined as meaning 262,143 repetitions. The compiler does check the syntax of strings (see section 3.9 on the efficient use of this facility).

A.4 Summary of Format Codes

A	alphabetic character represented as integer	X	arbitrary replicator
B	blank space	Z	zero suppression
C	comma	+	print the sign
D	digit	-	print the sign if it is minus
F	Boolean TRUE or FALSE	10	exponent part indicator
I	integer untranslated	()	delimiters of replicated format secondaries
L	Boolean untranslated	,	separates format items
P	Boolean bit	/	line alignment
R	real untranslated	↑	page alignment
S	string character	“ ”	delimiters of inserted string
T	truncation		
V	implied decimal point		decimal point

A.4. Summary of Format Codes. The following items have been added to format codes:

- J character alignment
- N standard format
- M variable of any type (octal representation)
- H integer variable representing up to 8 display code characters
- * page alignment

and C has been deleted.

A.5 "Standard" Format

There is a format available without specifications (cf. Section B.5) which has the following characteristics.

(a) On input, any number written according to the ALGOL syntax for $\langle \text{number} \rangle$ is accepted with the conventional meaning. These are of arbitrary length, and they are delimited at the right by the following conventions: (i) A letter or character other than a decimal point, sign, digit, or "10" is a delimiter. (ii) A sequence of k or more blank spaces serves as a delimiter as in (i); a sequence of less than k blank spaces is ignored. This number $k \geq 1$ is specified by the implementor (and the implementor may choose to let the programmer specify k on a control card of some sort). (iii) If the number contains a decimal point, sign, digit or "10" on the line where the number begins, the right-hand margin of that line serves as a delimiter of the number. However, if the first line of a field contains no such characters, the number is determined by reading several lines until finding a delimiter of type (i) or (ii). In other words, a number is not usually split across more than one line, unless its first line contains nothing but spaces or characters which do not enter into the number itself (see Section B.5 for further discussion of standard input format).

(b) On output, a number is given in the form of a decimal number with an exponent. This decimal number has the amount of significant figures which the machine can represent; it is suitable for reading by the standard input format. Standard output format takes a fixed number of characters on the output medium; this size is specified by each ALGOL installation. Standard output format can also be used for the output of strings, and in this case the number of characters is equal to the length of the string.

A.5 Standard Format

Standard format results from the format item N, from the exhaustion of the format string, and from an empty format string. The standard format for both integer and real variables is $+D.14D_{10}^3+D$. When the number cannot be written using the given format, the modified standard format used is $(^{*})D.14D_{10}^3+3D(^{*})$ (sections A.1.3.3 and $(^{*})$ A.1.3.7). The number of blank characters, k , serving as a delimiter between numbers in standard format may be specified by the channel statement (chapter 7). If none is specified, two is assumed. On both input and output the channel is left positioned immediately after the delimiter following the number read or written. String parameters can be output under the standard format nS , where n is the length of the string excluding the string quotes.

For an infinite or an indefinite value, two asterisks are printed for the value.

B. Input and Output Procedures

B.1 General Characteristics

The over-all approach to input and output which is provided by the procedures of this report will be introduced here by means of few examples, and the precise definition of the procedures will be given later.

Consider first a typical case, in which we want to print a line containing the values of the integer variables N and M, each of which is nonnegative, with at most five digits; also the value of $X[M]$, in the form of a signed number with a single nonzero digit to the left of the decimal point, and with an exponent indicated; and finally the value of $\cos(t)$ using a format with a fixed decimal point and no exponent. The following might be written for this case:

```
output 4(6, '2(BBBZZZZD) ,3B+ D.DDDDDD10+DDD,3B,
```

```
-Z.DDDDBDDDD/',N,M,X[M],cos(t)).
```

This example has the following significance. (a) The "4" in output 4 means four values are being output. (b) The "6" means that output is to go to unit 6.

This is the logical unit number, i.e., the programmer's number for that unit, and it does not necessarily mean physical unit number 6. See Section B.1.1, for further discussion of unit numbers. (c) The next parameter, '2(BBB. . . DDDD)', is the format string which specifies a format for outputting the four values. (d) The last four parameters are the values being printed. If $N = 500$, $M = 0$, $X[0] = 18061579$, and $t = 3.1415926536$, we obtain the line

```
□□□□□ 500 □□□□□□□□ 0 □□□□ +1.8061581 0+007 □□□□ -1.0000 □ 0000
```

as output.

Notice the "/" used in the above format; this symbol signifies the end of a line. If it had not been present, more numbers could have been placed on the same line in a future output statement. The programmer may build the contents of a line in several steps, as his algorithm proceeds, without automatically starting a new line each time output is called. For example, the above could have been written

```
output 1(6,'BBBZZZZD',N);
```

```
output 1(6,'BBBZZZZD',M);
```

```
output 2(6,'3B+D.DDDDDD10+DDD,3B,- Z.DDDDBDDDD', X[M],cos(t) );
```

```
output 0(6,'/');
```

with equivalent results.

In the example above a line of 48 characters was output. If for some reason these output statements are used with a device incapable of printing 48 characters on a single line, the output would actually have been recorded on two or more lines, according to a rule which automatically keeps from breaking numbers between two consecutive lines wherever possible. (The exact rule appears in Section B.5).

Now let us go to a slightly more complicated example:

the real array $A[1:n,1:n]$ is to be printed, starting on a new page. Supposing each element is printed with the format "BB-ZZZZ.DD", which uses ten characters per item, we could write the following program:

```
output 0(6,'/');
```

```
for i := 1 step 1 until n do
```

begin for j := 1 step 1 until n do output 1(6,'BB-ZZZZ.DD',

A[i,j]); output 0(6,'//') end.

If 10n characters will fit on one line, this little program will print n lines, double spaced, with n values per line; otherwise n groups of k lines separated by blank lines are produced, where k lines are necessary for the printing of n values. For example, if n = 10 and if the printer has 120 character positions, 10 double-spaced lines are produced. If, however, a 72-character printer is being used, 7 values are printed on the first line, 3 on the next, the third is blank, then 7 more values are printed, etc.

There is another way to achieve the above output and to obtain more control over the page format as well. The subject of page format will be discussed further in Section B.2, and we will indicate here the manner in which the above operation can be done conveniently using a single output statement. The procedures output 0, output 1, etc. mentioned above provide only for the common cases of output, and they are essentially a special abbreviation for certain calls on the more general procedure out list. This more general procedure could be used for the above problem in the following manner:

out list (6,LAYOUT,LIST)

Here LAYOUT and LIST are the names of procedures which appear below. The first parameter of out list is the logical unit number as described above. The second parameter is the name of a so-called "layout procedure"; general layout procedures are discussed in Section B.3. The third parameter of out list is the name of a so-called "list procedure"; general list procedures are discussed in Section B.4. In general, a layout procedure specifies the format control of the input or output. For the case we are considering, we could write a simple layout procedure (named "LAYOUT") as follows:

procedure LAYOUT; format 1(41,(X(BB-ZZZZ.DD),//)',n)

The 1 in format 1 means a format string containing one X is given.

The format string is 1.

(X(BB-ZZZZ.DD),//)

which means skip to a new page, then repeat the format X(BB-ZZZZ.DD),// until the last value is output. The latter format means that BB-ZZZZ.DD is to be used X times, then skip to a new line. Finally, format 1 is a procedure which effectively inserts the value of n for the letter X appearing in the format string.

A list procedure serves to specify a list of quantities. For the problem under consideration, we could write a simple list procedure (named "LIST") as follows:

procedure LIST(ITEM);for i := 1 step 1 until n do

for j := 1 step 1 until n do ITEM(A[i,j])

Here "ITEM A(i,j)" means that A(i,j) is the next item of the list. The procedure ITEM is a formal parameter which might have been given a different name such as PIECE or CHUNK; list procedures are discussed in more detail in Section B.4.

The declarations of LAYOUT and LIST above, together with the procedure statement out list (6,LAYOUT,LIST), accomplish the desired output of the array A.

Input is done in a manner dual to output, in such a way that it is the exact inverse of the output process wherever possible. The procedures in list and input n correspond to out list and output n ($n = 0, 1, \dots$). Two other procedures, get and put, are introduced to facilitate storage of intermediate data on external devices. For example, the statement put (100,LIST) would cause the values specified in the list procedure named LIST to be recorded in the external medium with an identification number of 100. The subsequent statement get (100,LIST) would restore these values. The external medium might be a disk file, a drum, a magnetic tape, etc.; the type of device and the format in which data is stored there is of no concern to the programmer.

B.1 General Characteristics. In order to be compiled without a diagnostic, the example above of procedure LIST must be written with specifications (chapter 2, section 5.4.5).

```
procedure LIST(ITEM); procedure ITEM;
```

```
  for i := 1 step 1 until n do
```

```
    for j := 1 step 1 until n do ITEM (A[i , j])
```

B.1.1 Unit numbers. The first parameter of input and output procedures is the logical unit number, i.e., some number which the programmer has chosen to identify some input or output device. The connection between logical unit numbers and the actual physical unit numbers is specified by the programmer outside of the ALGOL language, by means of "control cards" preceding or following his program, or in some other way provided by the ALGOL implementor. The situation which arises if the same physical unit is being used for two different logical numbers, or if the same physical unit is used both for input and for output, is undefined in general.

It is recommended that the internal computer memory (e.g. the core memory) be available as an "input-output device", so that data may be edited by means of input and output statements.

B.1.1 Unit Numbers. Wherever the term unit number appears in the ACM Report, channel number applies.

A channel is defined as all the specifications the input/output system needs to perform operations on a particular data file. A channel may be thought of as the set of descriptive information by which one reaches or knows of a data file. A channel number is the name of this set of descriptive information as well as the internal, indirect reference name of the data file accessed via this information.

The channel contains the following specifications about a data file:

1. Physical device description — device name, logical address, read or write mode
2. Status of physical device — device position, error conditions
3. Data file description — file name, read or write mode, blocking information
4. Data file status — file position, error conditions
5. Description of formatting area — buffer area from or to which data in a file is moved

Each channel is designated by a unique non-negative integer. If channel 0 is used, a channel statement defining it as indexed must be supplied. Channels are established by means of channel statements (chapter 7).

Channels are divided into four mutually incompatible sets according to the procedures that are available to transfer data:

1. Coded Sequential – INCHARACTER, OUTCHARACTER, INREAL, OUTREAL, INARRAY, OUTARRAY, INPUT, OUTPUT, INLIST, OUTLIST
2. Indexed – GETLIST, PUTLIST, GET, PUT, GETITEM, PUTITEM[†]
3. Word Addressable – FETCHLIST, STORELIST, STOREITEM, FETCHITEM
4. Binary Sequential – GETARRAY, PUTARRAY.

Any attempt to use a channel as a member of more than one set within a program will cause an error.

Input/output procedures for direct access devices (indexed and word addressable sets) are described in section 3.2.2 and those for binary sequential channels in section 3.2.3. Coded sequential channel characteristics and procedures are described below.

B.2 Horizontal and Vertical Control

This section deals with the way in which the sequence of characters, described by the rules of formats in Section A, is mapped onto input and output devices. This is done in a manner which is essentially independent of the device being used, in the sense that with these specifications the programmer can anticipate how the input or output data will appear on virtually any device. Some of the features of this description will, of course, be more appropriately used on certain devices than on others.

We will begin by assuming we are doing output to a printer. This is essentially the most difficult case to handle, and we will discuss the manner in which other devices fit into the same general framework. The page format is controlled by specifying the horizontal and the vertical layout. Horizontal layout is controlled essentially in the same manner as vertical layout, and this symmetry between the horizontal and vertical dimensions should be kept in mind for easier understanding of the concepts of this section.

Refer to figure 2, the horizontal format is described in terms of three parameters (L,R,P), and the vertical format has corresponding parameters (L',R',P'). The parameters L, L' and R, R' indicate left and right margins, respectively; Figure 2 shows a case where $L = L' = 4$ and $R = R' = 12$. Only position L through R of a horizontal line are used, and only lines L' and R' of the page are used; we require that $1 \leq L \leq R$ and $1 \leq L' \leq R'$. The parameter P is the number of characters per line, and P' is the number of lines per page. Although L, R, L' and R' are chosen by the programmer, the values of P and P' are characteristics of the device and they are usually out of the programmer's control. For those devices on which P and P' can vary (for example, some printers have two settings, one on which there are 66 lines per page, and another on which there are 88), the values are specified to the system in some manner external to the ALGOL program, e.g. on control cards. For certain devices, values P or P' might be essentially infinite.

[†]SCOPE 2 does not support indexed files.

Although Figure 2 shows a case where $P \geq R$ and $P' \geq R'$, it is of course quite possible that $P < R$ or $P' < R'$ (or both) might occur, since P and P' are in general unknown to the programmer. In such cases, the algorithm described in Section B.5 is used to break up logical lines which are too wide to fit on a physical line, and to break up logical pages which are too large to fit a physical page. On the other hand, the conditions $L \leq P$ and $L' \leq P'$ are insured by setting L or L' equal to 1 automatically if they happen to be greater than P or P' , respectively.

Characters determined by the output values are put onto a horizontal line; there are three conditions which cause a transfer to the next line: (a) normal line alignment, specified by a "/" in the format; (b) R-overflow, which occurs when a group of characters is to be transmitted which would pass position R ; and (c) P-overflow, which occurs when a group of characters is to be transmitted which would not cause R-overflow but would pass position P . When any of these three things occurs, control is transferred to a procedure specified by the programmer in case special action is desired (e.g. a change of margins in case of overflow; see Section B.3.3).

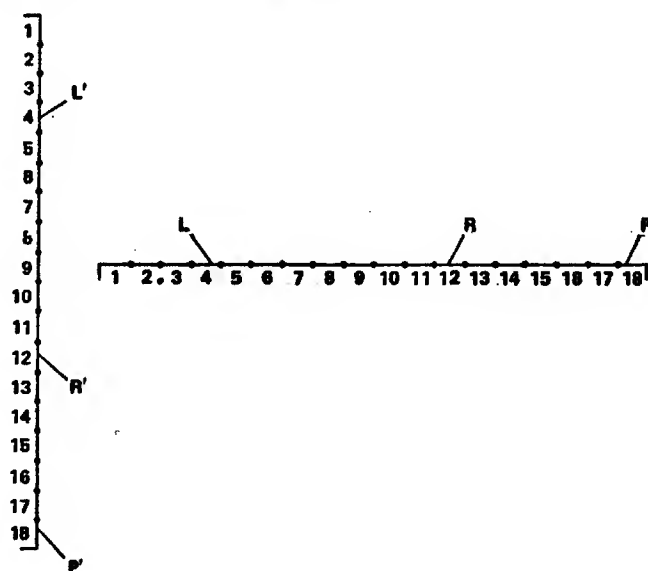


Figure 2.

Similarly, there are three conditions which cause a transfer to the next page: (a') normal page alignment, specified by a "↑" in the format; (b') R' -overflow, which occurs when a group of characters is to be transmitted which would appear on line $R'+1$; and (c') P' -overflow, which occurs when a group of characters is to be transmitted which would appear on line $P'+1 < R'+1$. The programmer may indicate special procedures to be executed at this time if he wishes, e.g. to insert a page heading, etc.

Further details concerning pages and lines will be given later. Now we will consider how devices other than printers can be thought of in terms of the ideas above.

A typewriter is, of course, very much like a printer and it requires no further comment.

Punched cards with, say, 80 columns, have $P = 80$ and $P' = \infty$. Vertical control would appear to have little meaning for punched cards, although the implementor might choose to interpret "↑" to mean the insertion of a coded or blank card.

With paper tape, we might again say that vertical control has little or no meaning; in this case, P could be the number of characters read or written at a time.

On magnetic tape capable of writing arbitrarily long blocks, we have $P = P' = \infty$. We might think of each page as being a "record", i.e., an amount of contiguous information on the tape which is read or written at once. The lines are subdivisions of a record, and R' lines form a record; R characters are in each line. In this way we can specify so-called "blocking of record." Other interpretations might be more appropriate for magnetic tapes at certain installations, e.g. a format which would correspond exactly to printer format for future offline listing, etc.

These examples are given merely to indicate how the concepts described above for printers can be applied to other devices. Each implementor will decide what method is most appropriate for his particular devices, and if there are choices to be made they can be given by the programmer by means of control cards. The manner in which this is done is of no concern in this report; our procedures are defined solely in terms of P and P' .

B.2 Horizontal and Vertical Control

The values of P and P' are specified to the system by a suitable call to the procedure SYSPARAM (section B.6) or with channel statements.

The initial value of P on the channel statement (chapter 7) defines the maximum size of the line to be read or written. P may be changed during program execution, but it must never exceed its initial setting. The initial value of P' on the channel statement defines the number of lines per page; the value of this parameter may be changed to exceed its initial setting.

If the initial value of P' is zero, it cannot be changed to a nonzero value. If the initial value of P' is nonzero, then it can be changed to any nonzero value, even one exceeding the initial value.

There are two kinds of coded sequential channels, paged and unpagged. Paged channels are those that can be printed. They must have $P' \neq 0$; " \uparrow " and "/" have their normal meaning.

Unpagged channels must have $P' = 0$; the value of p' is always 0. " \uparrow " and "/" have no special meaning and result in a line alignment for a nonempty line, and no alignment when the current line is empty. Horizontal control, as discussed throughout this chapter, is applicable to unpagged channels, but vertical control has no meaning.

B.3 Layout Procedures

Whenever input or output is done, certain "standard" operations are assumed to take place, unless otherwise specified by the programmer. Therefore one of the parameters of the input or output procedure is a so-called "layout" procedure, which specifies all of the nonstandard operations desired. This is achieved by using any or all of the six "descriptive procedures" format, h end, v end, h lim, v lim, no data described in this section.

The precise action of these procedures can be described in terms of the mythical concept of six "hidden variables," $H1, H2, H3, H4, H5, H6$. The effect of each descriptive procedure is to set one of these variables to a certain value; and as a matter of fact, that may be regarded as the sum total of the effect of a descriptive procedure. The programmer normally has no other access to these hidden variables (see, however, Section B.7). The hidden variables have a scope which is local to in list and to out list.

B.3 Layout Procedures

A seventh descriptive procedure, TABULATION, has been added with a corresponding hidden value $H6$ (Section B.3.3). The hidden variable corresponding to NO DATA is $H7$ rather than $H6$.

If any of the procedures `FORMAT`, `H END`, `V END`, `H LIM`, `V LIM`, `TABULATION`, or `NO DATA` are called when neither `IN LIST` nor `OUT LIST` is active, they have the effect of a dummy procedure; a procedure call is made and the procedure is exited immediately.

B.3.1 Format Procedures. The descriptive procedure call

`format (string)`

has the effect of setting the hidden variable `H1` to indicate the string parameter. This parameter may either be a string explicitly written, or a formal parameter; but in any event, the string it refers to must be a format string, which satisfies the syntax of Section A.3, and it must have no "X" replicators.

The procedure `format` is just one of a class of procedures which have the names `format n` ($n = 0, 1, \dots$). The name `format` is equivalent to `format 0`. In general, the procedure `format n` is used with format strings which have exactly n X-replicators. The call is

`format n (string, X_1, X_2, \dots, X_n)`

where each X_i is an integer parameter called by value. The effect is to replace each X of the format string by one of the X_i , with the correspondence defined from left to right. Each X_i must be nonnegative.

For example,

`format 2 ('XB . XD 10+DD',5,10)`

is equivalent to

`format ('5B . 10D10+DD').`

B.3.1 Format Procedures. The single procedure with call:

`FORMAT (string, X_1, X_2, \dots, X_n)`

replaces the $n+1$ procedures defined in the proposal with call:

`FORMAT n (string, X_1, X_2, \dots, X_n)`

The number of parameters included in the call to `FORMAT` defines the equivalent procedure defined in the proposal. For example:

`FORMAT (string, X_1, X_2)` is equivalent to

`FORMAT 2 (string, X_1, X_2)` as defined in the proposal.

The number of X replicators is limited only by the fact that there is a maximum of 63 actual parameters in a procedure call.

If the format string in a format call contains n X replicators, it must be followed by at least n nonnegative integer parameters. If the number of nonnegative integer parameters exceeds the number of X replicators, then the n X replicators in the format string will be replaced by the first n parameters; however, if there are fewer integer parameters than required by the format string, an error occurs.

B.3.2 Limits. The descriptive procedure call

h lim (L,R)

has the effect of setting the hidden variable H2 to indicate the two parameters L and R. Similarly,

v lim (L',R')

sets H3 to indicate L' and R'. These parameters have the significance described in Section B.2. If h lim and v lim are not used, $L = L' = 1$ and $R = R' = \infty$.

B.3.2 Limits. Any attempt to set $R < L$ or $R' < L'$ will be ignored. If $L > P$, the value 1 is substituted for L; similarly, if $L' > P'$, the value 1 is substituted for L'.

B.3.3 End Control. The descriptive procedure

h end (P_N,P_R,P_P); v end (P_{N'},P_{R'},P_{P'})

have the effect of setting the hidden variables H4 and H5, respectively, to indicate their parameters. The parameters P_N,P_R,P_P, P_{N'},P_{R'},P_{P'}, are names of procedures (ordinally dummy statements if h end and v end are not specified) which are activated in case of normal line-alignment, R-overflow, P-overflow, normal page alignment, R'-overflow and P'-overflow, respectively.

B.3.3 End Control. The descriptive procedure call

TABULATION (n)

sets the hidden variable H6 to indicate the parameter n. Here n is the width of the tabulation field, measured in the number of characters on the external device (section B.5.1, process C). If TABULATION is not called then H6 = 1.

Tabulation is controlled by a J in the format. This causes the character pointer to be advanced to the next tabulation position with intermediate positions being filled with blanks.

If the tabulation spacing is N, then the first character of the tabulation fields would be:

$L, L+N, L+2N, \dots, L+KN$ where $L+KN \leq \min(P, R)$.

B.3.4 End of Data. The descriptive procedure call

no data (L)

has the effect of setting the hidden variable H6 to indicate the parameter L. Here L is a label. End of data as defined here has meaning only on input, and it does not refer to any specific hardware features; it occurs when data is requested for input but no more data remains on the corresponding input medium. At this point, a transfer to statement labelled L will occur. If the procedure no data is used, transfer will occur to a "label" which has effectively been inserted just before the final end in the ALGOL program, thus terminating the program. (In this case the implementor may elect to provide an appropriate error comment).

B.3.4 End of Data. The call to NO DATA sets the hidden variable H7 rather than H6. End of data is defined as the occurrence of an end of partition condition on the input device (end of section on the file INPUT).

If the procedure NO DATA was not called, transfer occurs to the label established for the channel by the EOF or CHANERROR procedure (Section 3.3). If neither of these procedures was called or if an established label is no longer accessible, the object program terminates abnormally with the message UNCHECKED EOF.

B.3.5 Examples. A layout procedure might look as follows:

```
procedure LAYOUT; begin format ('/');  
  
    if B then begin format (1('XB',Y + 10); no data (L32) end;  
  
    h lim (if B then 1 else 10,30) end;
```

Note that layout procedures never have formal parameters; this procedure, for example, refers to three global quantities, B, Y and L32. Suppose Y has the value 3; then this layout accomplishes the following:

<u>Hidden Variable</u>	<u>Procedure</u>	<u>if B = true</u>	<u>if B = false</u>
H1	format	'13B'	'/'
H2	h lim	(1,30)	(10,30)
H3	v lim	(1,∞)	(1,∞)
H4	h end	(, ,)	(, ,)
H5	v end	(, ,)	(, ,)
H6 [†]	tabulation	1	1
H7	no data	L32	end program

As a more useful example, we can take the procedure LAYOUT of Section B.1 and rewrite it so that the horizontal margins (11,110) are used on the page, except that if P-overflow or R-overflow occurs we wish to use the margins (16,105) for overflow lines.

```
procedure LAYOUT; begin  
  
    format 1 ('^(X(BB-ZZZZ.DD) ,//)',n);  
  
    h lim (11,110); h end (K,L,L) end;  
  
procedure K; h lim (11,110);  
  
procedure L; h lim (16,105);
```

[†]This definition replaces that in the ACM Report.

This causes the limits (16,105) to be set whenever overflow occurs, and the "/" in the format will reinstate the original margins when it causes procedure K to be called. (If the programmer wishes a more elaborate treatment of the overflow case, depending on the value of P, he may do this using the procedures of Section B.6).

B.4 List Procedures

B.4.1 General characteristics. The concept of a list procedure is quite important to the input-output conventions described in this report, and it may also prove useful in other applications of ALGOL. It represents a specialized application of the standard features of ALGOL which permit a procedure identifier, L, to be given as an actual parameter of a procedure, and which permit procedures to be declared within procedures. The purpose of a list procedure is to describe a sequence of items which is to be transmitted for input or output. A procedure is written in which the name of each item V is written as the argument of a procedure, say ITEM, thus: ITEM(V). When the list procedure is called by an input-output system procedure, another procedure (such as the internal system procedure out item) will be "substituted" for ITEM, V will be called by name, and the value of V will be transmitted for input or output. The standard sequencing of ALGOL statements in the body of the list procedure determines the sequence of items in the list.

A simple form of list procedure might be written as follows:

```
procedure LIST (ITEM);  
  
begin ITEM(A); ITEM(B); ITEM(C) end
```

which says that the values of A, B, and C are to be transmitted.

A more typical list procedure might be:

```
procedure PAIRS (ELT);  
  
for i := 1 step 1 until n do begin ELT(A[i]);  
  
    ELT(B[i]) end
```

This procedure says that the values of the list of items A[1], B[1], A[2], B[2], . . . , A[n], B[n] are to be transmitted, in that order. Note that if $n \leq 0$ no items are transmitted at all.

The parameter of the "item" procedure (i.e., the parameter of ITEM or ELT in the above examples) is called by name. It may be an arithmetic expression, a Boolean expression, or a string, in accordance with the format which will be associated with the item. Any of the ordinary features of ALGOL may be used in a list procedure, so there is great flexibility.

Unlike layout procedures which simply run through their statements and set up hidden variables H1 through H6, a list procedure is executed step by step with the input or output procedure, with control transferring back and forth. This is accomplished by special system procedures such as in item and out item which are "interlaced" with the list procedure, as described in Sections B.4.2 and B.5. The list procedure is called with in item (or out item) as actual parameter, and whenever this procedure is called within the list procedure, the actual input or output is taking place. Through the interlacing, special format control, including the important device-independent overflow procedures, can take place during the transmission process. Note that a list procedure may change the hidden variables by calling a descriptive procedure; this can be a valuable characteristic, e.g. when changing the format, based on the value of the first item which is input.

B.4.1 General characteristics. List procedures are used in conjunction with the list-driven input/output procedures to describe the sequence of items which is to be transmitted for input or output. A list procedure has one parameter which is itself a procedure, for example, named ITEM. Each input/output item is written as a parameter to procedure ITEM, for example, ITEM (u). When the list procedure is called by an input/output system procedure, the internal system procedure INITEM or OUTITEM is effectively 'substituted' for ITEM, u is called by name and the value of u is transmitted for input or output.

In order to be compiled without a diagnostic, the above two examples must be written with specifications (chapter 2, section 5.4.5).

```
procedure LIST (ITEM); procedure ITEM;
begin ITEM (A); ITEM (B); ITEM (C) end

procedure PAIRS (ELT); procedure ELT;
for i:= 1 step 1 until n do begin ELT (A[i]);
    ELT (B[i]) end
```

B.4.1.1 Additional parameter type. An array name may be a parameter to the following procedures: OUTPUT, INPUT, GETITEM, PUTITEM, FETCHITEM, STOREITEM. A call to one of these procedures with an array parameter is equivalent to a sequence of calls with each of the subscripted variables of the array as parameters in lexicographical order, i.e., with last subscript varying fastest. Each subscripted variable uses one format item.

B.4.2 Other applications. List procedures can actually be used in many ways in ALGOL besides their use with input or output routines; they are useful for manipulating linear lists of items of a quite general nature. To illustrate this fact, and to point out how the interlacing of control between list and driver procedures can be accomplished, here is an example of a procedure which calculates the sum of all of the elements in a list (assuming all elements are of integer or real type):

```
procedure ADD(Y,Z); begin

procedure A(X); Z := Z+X

Z := 0; Y(A) end
```

The call ADD (PAIRS,SUM) will set the value of SUM to be the sum of all of the items in the list PAIRS defined in Section B.4.1. The reader should study this example carefully to grasp the essential significance of list procedures. It is a simple and instructive exercise to write a procedure which sets all elements of a list to zero.

B.5 Input and Output Calls

Here procedures are described which cause the actual transmission of input or output to take place.

B.5 Input and Output Calls

To give a more complete range of input/output procedures, the following calls have been added:

INCHARACTER	INREAL	INARRAY
OUTCHARACTER	OUTREAL	OUTARRAY
CHLENGTH		

Character Transmission

The procedures INCHARACTER and OUTCHARACTER provide the means of communicating between input/output devices and the variables of the program in terms of basic characters. (The basic characters are given in appendix C.) The procedure calls are as follows:

INCHARACTER (channel, string, destination)

OUTCHARACTER (channel, string, source)

Where channel and source must be arithmetic expressions called by value, string is a string, called by name, and destination is an integer variable called by name.

The corresponding procedure declarations might be defined as:

```
procedure INCHARACTER (channel, string, destination); value channel;  
             integer channel, destination; string string;  
             < procedure body >
```

```
procedure OUTCHARACTER (channel, string, source); value channel, source;  
             integer channel, source; string string;  
             < procedure body >
```

In both procedures, the correspondence between the basic characters on the input/output device and the value of the source or destination in the program is established by mapping the sequence of basic characters given in the string (second parameter), taken from left to right, onto the positive integers 1, 2, 3, . . . Using this correspondence, the procedure INCHARACTER assigns to the integer variable given as the third parameter the value corresponding to the next basic character appearing on the input stream. If this next basic character does not appear in the string parameter, the number 0 is assigned. If the next character appearing in the input stream is the end-of-line symbol, the value -1 is assigned.

For example, the procedure call INCHARACTER (61, 'ABC', i) will set i = 1 if the next character on the input device is A, i = 2 if B, i = 3 if C, i = 0 if a basic character other than A, B and C, -1 if end-of-line symbol.

Similarly, the procedure OUTCHARACTER transfers the basic character corresponding to the third parameter to the output device. If the value of this third parameter is -1, the end-of-line symbol is transferred.

An integer procedure, CHLENGTH, calculates the length of an actual or formal string. The procedure call is:

CHLENGTH (string)

and the procedure declaration is:

integer procedure CHLENGTH (string); string string; <procedure body>

The value of CHLENGTH (string) is the number of basic characters of the open string enclosed between the outermost string quotes.

Type Real Transmission

Transmission of information of type real between variables of the program and an external device may be accomplished by the procedure calls

INREAL (channel, destination)

OUTREAL (channel, source)

where channel and source are arithmetic expressions called by value and destination is a variable of type real called by name, as follows:

procedure INREAL (channel, destination); value channel; integer channel;
real destination;
<procedure body>

procedure OUTREAL (channel, source); value channel, source; integer channel;
real source;
<procedure body>

The two procedures INREAL and OUTREAL form a pair. The procedure INREAL assigns the next value appearing on the input device to the real variable given as the second parameter. Similarly, procedure OUTREAL transfers the value of the second actual parameter to the output device.

A value which has been transferred by the call OUTREAL is represented in such a way that the same value, in the sense of numerical analysis, can be transferred back to a variable by means of procedure INREAL.

The procedures INREAL and OUTREAL handle numbers in standard format (cf. section A.5).

Array Transmission

Arrays may be transferred between input/output devices by means of the procedure calls

INARRAY (channel, destination)

OUTARRAY (channel, source)

where channel is an arithmetic expression and destination and source are arrays of type real, as follows:

Procedure INARRAY (channel, destination); value channel; integer channel;
array destination;
 <procedure body>

Procedure OUTARRAY (channel, source); value channel; integer channel;
array source;
 <procedure body>

Procedures INARRAY and OUTARRAY also form a pair; they transfer the ordered set of values that form the array given as the second parameter. The array bounds are defined by the corresponding array declaration rather than by additional parameters (the mechanism for doing this is already available in ALGOL for the value call of arrays).

The order in which the elements of the array are transferred corresponds to the lexicographic order of the values of the subscripts as follows:

$a(k_1, k_2, \dots, k_m)$ precedes

$a(j_1, j_2, \dots, j_m)$ provided

$k_i = j_i$ ($i = 1, 2, \dots, p-1$)

and $k_p < j_p$ ($1 \leq p \leq m$)

That is, an array is stored by rows.

Any multidimensional structure of the array is not reflected in the corresponding numbers on the external device which appear only as a linear sequence.

The representation of the numbers on the external device conforms to the same rules as given for INREAL and OUTREAL; it is possible to input numbers by INREAL that have previously been output by OUTARRAY.

Examples:

The following examples may help to clarify the uses of the procedures described above:

- (a) procedure outboolean (channel, boolean); value boolean; integer channel; Boolean boolean; comment this procedure outputs a Boolean value as the basic symbol true or false; begin integer i;
if boolean then begin for i : = 1 step 1 until 7 do
 OUTCHARACTER (channel, ' ' TRUE ' ' , i)
end
else for i : = 1 step 1 until 7 do
 OUTCHARACTER (channel, ' ' FALSE ' ' , i)
end
- (b) procedure outstring (channel, string); value channel; integer channel; string string; comment outputs the string to the output device;
begin integer i;
 for i : = 1 step 1 until CHLENGTH (string)
 do OUTCHARACTER (channel, string, i)
end
- (c) procedure integer (channel, integer); value channel; integer channel, integer; comment inputs an integer which, on the input device, appears as a sequence of digits, possibly preceded by a sign and followed by a comma. Any symbol in front of the sign is discarded;

begin integer n,k; Boolean b ;

integer : = 0 ; b : = true;

for k : = 1, k+1 while n = 0 do INCHARACTER (channel, '0123456789-+', n);

if n = 11 then b : = false ;

if n > 10 then n : = 1 ;

for k : = 1, k+1 while n ≠ 13 do

begin integer : = 10 * integer + n - 1;

 INCHARACTER (channel, '0123456789-+', n)

end k ;

if ¬ b then integer : = - integer

end

(d) begin begin array a [1:10] ; < statements > ;

OUTARRAY (15, a)

end

begin array b [0 : 1, 1 : 5] ; INARRAY (15, b);

< statements >

end

end

- (e) the following example exhibits the use of INARRAY and OUTARRAY for inversion of a matrix including transfer of the matrix elements from and to the external device. It requires that an appropriate declaration for a matrix inversion procedure as well as the declaration of outstring as given above in example b are inserted at appropriate places in the program.

begin integer n; INREAL (5, n) ; comment the matrix elements must be preceded by the order;

begin array a [1 : n , 1 : n] ; INARRAY (5, a);

matrix inversion (n, a, singular) ; OUTARRAY (15, a) ;

go to Ex

end;

singular : outstring (15, 'singular');

Ex : end

B.5.1 Output

An output process is initiated by the call:

out list (unit,LAYOUT,LIST)

Here unit is an integer parameter called by value, which is the number of an output device (cf. Section B.1.1). The parameter LAYOUT is the name of a layout procedure (Section B.3) and LIST is the name of a list procedure (Section B.4).

There is also another class of procedures, named output n, for $n = 0, 1, 2, \dots$, which is used for output as follows:

output n (unit, format string, e_1, e_2, \dots, e_n)

B.5.1 Output. The single procedure with call:

OUTPUT (channel, format string, e_1, e_2, \dots, e_n)

replaces the $n+1$ procedures defined with call:

OUTPUT n (channel, format string, e_1, e_2, \dots, e_n)

$n = 0, \infty$

The number of variables included in the call to OUTPUT defines which of the $n+1$ procedures (defined in the ACM Report) it is equivalent to. For example:

OUTPUT (channel, format string, e_1)

is equivalent to

OUTPUT 1 (channel, format string, e_1)

defined in the report.

The parameter e_1 may be any of the types described in section B.4.1.1.

For example:

OUTPUT (61, '/', + 3ZD.2D',A)

OUTPUT (61, '3 (N), 'TOTAL' ', X, Y, Z)

Each of these latter procedures can be defined in terms of out list as follows:

procedure output n (unit, format string, e_1, e_2, \dots, e_n)

begin procedure A; format (format string);

procedure B(P); begin P(e_1); P(e_2); \dots ; P(e_n) end;

out list (unit, A,B) end

The procedure OUTPUT is defined in terms of OUTLIST as follows:

procedure OUTPUT (channel, format string, e_1, e_2, \dots, e_n);

value channel; integer channel; string format string; comment e_1, e_2, \dots, e_n may be a string, any integer, real or Boolean expression, or any integer, real or Boolean array;

begin procedure a; FORMAT (format string);

procedure b (p); procedure p ;

begin integer i;
for i : = 1 step 1 until n do p (e_i)
end;

OUTLIST (channel, a, b)

end

We will therefore assume in the following rules that out list has been called.

Let the variables p and p' indicate the current position in the output for the unit under consideration, i.e., lines $1, 2, \dots, p'$ of the current page have been completed, as well as character positions $1, 2, \dots, p$ of the current line (i.e., of line $p'+1$). At the beginning of the program, $p = p' = 0$. The symbols P and P' denote the line size and page size (see Section B.2). Output takes place according to the following algorithm:

Step 1. The hidden variables are set to standard values:

H1 is set to the "standard" format ' ' .

H2 is set so that $L = 1, R = \infty$

H3 is set so that $L' = 1, R' = \infty$

H4 is set so that P_N, P_R, P_P are all effectively equal to the DUMMY procedures defined as follows:
"procedure DUMMY;".

H5 is set so that P_N, P_R, P_P are all effectively equal to DUMMY.

H6 is set to terminate the program in case the data ends (this has meaning only on input).

Step 2. The layout procedure is called; this may change some of the variables H1, H2, H3, H4, H5, H6.

Step 3. The next format item of the format string is examined. (Note. After the format string is exhausted, "standard" format, Section A.5, is used from then on until the end of the procedure. In particular, if the format string is "", standard format is used throughout.) Now if the next format item is a title format, i.e., requires no data item, we proceed directly to step 4. Otherwise, the list procedure is activated; this is done the first time by calling the list procedure, using as actual parameter a procedure named out item; this is done on all subsequent times by merely returning from the procedure out item, which will cause the list procedure to be continued from the latest out item call. (Note: The identifier out item has scope local to out list, so a programmer may not call this procedure directly). After the list procedure has been activated in this way, it will either terminate or will call the procedure out item. In the former case, the output process is completed; in the latter case, continue at step 4.

Step 4. Take the next item from the format string. (Notes. If the list procedure was called in step 3, it may have called the descriptive procedure format, thereby changing from the format which was examined during step 3. In such a case, the new format is used here. But at this point the format item is effectively removed from the format string and copied elsewhere so that the format string itself, possibly changed by further calls of format, will not be interrogated until the next occurrence of step 3. If the list procedure has substituted a title format for a nontitle format, the "item" it specifies will not be output, since a title format consists entirely of insertions and alignment marks.)

Set "toggle" to false. (This is used to control the breaking of entries between lines.) The alignment marks, if any, at the left of the format item, now cause process A (below) to be executed for each "/", and process B for each "↑". If the format item consists entirely of alignment marks, then go immediately to step 3. Otherwise the size of the format (i.e., the number of characters specified in the output medium) is determined. Let this size be denoted by S . Continue with step 5.

Step 5. Execute process C, to ensure proper page alignment.

Step 6. Line alignment: if $\rho < L - 1$, effectively insert blank spaces so that $\rho = L - 1$. Now if $\text{toggle} = \text{true}$, go to step 9; otherwise, test for line overflow as follows: If $\rho + S > R$, perform process D, then call P_R and go to step 8; otherwise, if $\rho + S > P$, perform process D, call P_P , and go to step 8.

Step 7. Evaluate the next output item and output it according to the rules given in Section A; in the case of a title format, this is simply a transmission of the insertions without the evaluation of an output item. The pointer ρ is set to $\rho + S$. Any alignment marks at the right of the format item now cause activation of process A for each "/" and of process B for each "↑". Return to step 3.

Step 8. Set toggle to true. Prepare a formatted output item as in step 7, but do not record it on the output medium yet (this is done in step 9). Go to step 5. (It is necessary to re-examine page and line alignment, which may have been altered by the overflow procedure; hence we go to step 5 rather than proceeding immediately to step 9.)

Step 9. Transfer as many characters of the current output item as possible into positions $\rho + 1, \dots$, without exceeding position P or R. Adjust ρ appropriately. If the output of this item is still unfinished, execute process D again, call P_R (if $R \leq P$) or P_P (if $P < R$), and return to step 5. The entire item will eventually be output, and then we process alignment characters as in step 7, finally returning to step 3.

Process A. ("/" operation) Check page alignment with process C, then execute process D and call procedure P_N .

Process B. ("↑" operation) If $\rho > 0$, execute process A. Then execute process E and call procedure P_N .

Process C. (Page alignment)

If $\rho' < L' - 1$ and $\rho > 0$: execute process D, call procedure P_N , and repeat process C.

If $\rho' < L' - 1$ and $\rho = 0$: execute process D until $\rho' = L' - 1$.

If $\rho' + 1 > R'$: execute process E, call procedure P_R , and repeat process C.

If $\rho' + 1 > P'$: execute process E, call procedure P_P , and repeat process C.

Process D. Skip the output medium to the next line, set $\rho = 0$, and set $\rho' = \rho' + 1$.

Process E. Skip the output medium to the next page, and set $\rho' = 0$.

Steps 1–9 and processes A–E have been implemented as follows:

Step 1. (Initialization)

The hidden variables are set to standard values:

H1 is set to the standard format ' '.

H2 is set so that $L = 1, R = \infty$.

H3 is set so that $L' = 1, R' = \infty$.

H4 is set so that P_N, P_R, P_P are all effectively equal to the DUMMY procedure defined as follows:
procedure DUMMY ; ;

H5 is set so that P_N, P_R, P_P are all effectively equal to DUMMY.

H6 is set so that $TAB = 1$.

Step 2. (Layout)

The layout procedure is called; some of the variables H1, H2, H3, H4, H5, H6 might be changed. T is set to false. T is a Boolean variable used to control the sequencing of data with respect to title formats; T = true means a value has been transmitted to the procedure but has not yet been output.

Step 3. (Communication with list procedure)

The next format item of the format string is examined. (After the format string is exhausted, standard format is used until the end of the procedure. In particular, if the format string is ' ', standard format is used throughout.) If the next format item is a title format (requires no data item), proceed directly to step 4. If T = true proceed to step 4. Otherwise, the list procedure is called, using a procedure named OUTITEM as the actual parameter. Each subsequent return from OUTITEM causes the list procedure to be continued from the latest OUTITEM call. Since the scope of the identifier OUTITEM is local to OUTLIST, OUTITEM cannot be called directly.

After the list procedure has been activated it either terminates or calls OUTITEM. If it terminates, the output process is complete. If OUTITEM is called, T is set to true and any assignments to hidden variables that may have been made by calls to list procedures cause adjustment to the variables H1, H2, H3, H4, H5, H6, which are local to OUTITEM. The procedure then continues at step 4.

Step 4. (Alignment marks)

If the next format item includes an alignment mark it is removed from the format string and process A (a sub-routine below) is executed for each /, process B for each ↑ and process C for each J. Overflow procedures might cause the format strings to be changed. In this case, the new format string is not examined until step 6.

Step 5. (Get within margins)

Process G is executed to ensure proper page and line alignment.

Step 6. (Formatting the output)

The next item is taken from the format string.

In unusual cases, the list procedure or an overflow procedure may have called the descriptive procedure FORMAT, thereby changing the format string. If so, the new format string is examined from the beginning; it is conceivable that the format items examined in steps 3, 4, 6 might be three different formats. At this point, the current format item is effectively removed from the format string and copied elsewhere so that the format string itself, possibly changed by further calls to FORMAT, is not examined until the next occurrence of step 3.

Alignment marks at the left of the format item are ignored. If the format item is not composed only of alignment marks and insertions, the value of T is examined. If T = false, action is undefined (a nontitle format has been substituted for a title format in an overflow procedure, which is not allowed). Otherwise, the output item is evaluated and T is set to false. The format rules are applied and the characters $X_1X_2\ldots X_S$, which represent the formatted output on the external device, are determined. The number of characters, s, might depend upon the value being output using A, H or S format, as well as on the output medium.

Step 7. (Check for overflow)

If $\rho + s \leq R$ and $\rho + s \leq P$, where s is the size of the item as determined in step 6, the item fits on this line, so step 9 is executed. Otherwise, if the present item uses A, H or S format or title sequences a special nonbasic symbol is output to ensure that input is inverse to output. Continue at step 8.

Step 8. (Processing of overflow)

Process H ($\rho + s$) is performed. If $\rho + s \leq R$ and $\rho + s \leq P$, step 9 is executed; otherwise k is set to $\min(R, P) - \rho$. $x_1 x_2 \dots x_k$ are output, ρ is set to $\min(R, P)$, and $x_1 x_2 \dots x_{s-k}$ to $x_{k+1} x_{k+2} \dots x_s$. s is decreased by k and step 8 repeated.

Step 9. (Finish the item)

$x_1 x_2 \dots x_s$ are output, and ρ increased by s . Any alignment marks at the right of the format item now cause activation of process A for each /, process B for each ↑, and process C for each J. Return to step 3.

Process A (/ operation)

Page alignment is checked with process F, and process D is executed. Procedure P_N is called.

Process B (↑ operation)

If $\rho > 0$, process D is executed and procedure P_N is called. Process E is then executed and procedure $P_{N'}$ called.

Process C (J operation)

Page and line alignment are checked with process G. Then k is set to $((\rho - L + 1) \div \text{TAB} + 1) \times \text{TAB} + L - 1$ (the next tab setting for ρ), where TAB is the tab spacing for this channel. If $k \geq \min(R, P)$, process H(k) is performed; otherwise blanks are inserted until $\rho = k$.

Process D (New line)

The output device is skipped to the next line, ρ is set to 0, and ρ' is set to $\rho' + 1$.

Process E (New page)

The output device is skipped to the next page, and ρ is set to 0.

Process F (Page alignment)

If $\rho' + 1 < L'$ process D is executed until $\rho' = L' - 1$. If $\rho' + 1 > R'$, process E is executed, $P_{R'}$ is called, and process F is repeated. If $\rho' + 1 > P'$, process E is executed, $P_{P'}$ is called, and process F is repeated. Process F must terminate because $1 \leq L' \leq R'$ and $1 \leq L' \leq P'$. If a value $L' > P$ is chosen, L' is set to 1.

Process G (Page and line alignment)

Process F is executed. Then, if $\rho + 1 < L$, blank spaces are output until $\rho + 1 = L$. If $\rho + 1 > R$ or $\rho + 1 > P$, process H ($\rho + 1$) is performed. This process must terminate because $1 \leq L \leq R$ and $1 \leq L \leq P$. (If a value of $L > P$ is chosen, L is set to 1.)

Process H(k) (Line overflow)

Process D is performed. If $k > R$, P_R is called; otherwise P_P is called. Then process G is performed to ensure page and line alignment. Upon return from any of the overflow procedures, any assignments to hidden variables made by calls to descriptive procedures cause adjustment to the corresponding variables, local to OUTITEM, H1, H2, H3, H4, H5, and H6.

B.5.2 Input

The input process is initiated by the call:

in list (unit, LAYOUT, LIST)

The parameters have the same significance as they did in the case of output, except that unit is in this case the number of an input device. There is a class of procedures input n which stand for a call with a particularly simple type of layout and list, just as discussed in Section B.5.1 for the case of output. In the case of input, the parameters of the "item" procedure within the list must be variables.

B.5.2 Input. The single procedure with call:

INPUT (channel, format string, e_1, e_2, \dots, e_n)

replaces the n+1 procedures defined in the ACM Report with call:

INPUT n (channel, format string, e_1, e_2, \dots, e_n)

$n = 0, \infty$

The number of variables included in the call to INPUT defines which of the n+1 procedures (defined in the report) it is equivalent to. For example:

INPUT (channel, format string, e_1, e_2, e_3)

is equivalent to

INPUT 3 (channel, format string, e_1, e_2, e_3)

defined in the proposal.

A call to INPUT may include 0-61 e parameters.

The parameter e_i may be any of the types described in section B.4.1.1.

The procedure INPUT may be defined in terms of INLIST as follows:

```

procedure INPUT (channel, format string,  $e_1, e_2, \dots, e_n$ );
  value channel; integer channel; string format string;

  comment  $e_1, e_2, \dots, e_n$  may be any integer, real or Boolean variable, or any integer, real or Boolean array;

  begin procedure a; FORMAT (format string);

    procedure b (p); procedure p ;

      begin integer i;
        for i : = 1 step 1 until n do p ( $e_i$ )

      end;

    INLIST (channel, a, b)

  end

```

The various steps which take place during the execution of in list are very much the same as those in the case of out list, with obvious changes. Instead of transferring characters of title format, the characters are ignored on input. If the data is improper, some standard error procedure is used. (Cf. Section A.1.3.8.)

The only significant change occurs in the case of standard input format, in which the number S of the above algorithm cannot be determined in step 4. The tests $\rho + S > R$ and $\rho + S > P$ now become a test on whether positions $\rho + 1, \rho + 2, \dots, \min(R, P)$ have any numbers in them or not. If so, the first number, up to its delimiter, is used; the R and P positions serve as delimiters here. If not, however, overflow occurs, and subsequent lines are searched until a number is found (possibly causing additional overflows). The right boundary $\min(R, P)$ will not count as a delimiter in the case of overflow. This rule has been made so that the process of input is dual to that of output: an input item is not split across more than one line unless it has overflowed twice. Notice that the programmer has the ability to determine the presence or absence of data on a card when using standard format, because of the way overflow is defined. The following program, for example, will count the number n of data items on a single input card and will read them into A[1], A[2], ..., A[n]. (Assume unit 5 is a card reader.)

```

procedure LAY; h end (EXIT,EXIT,EXIT);

procedure LIST (ITEM); ITEM (A[n + 1] );

procedure EXIT; go to L2;

N : = 0; L1: in list (5,LAY,LIST); n : = n + 1; go to L1;

L2 :; comment mission accomplished;

```

Steps 1-9 and processes A-E of input have been implemented as follows:

Step 1. (Initialization)

The hidden variables are set to standard values:

H1 is set to the standard format ' '.

H2 is set so that $L = 1$, $R = \infty$.

H3 is set so that $L' = 1$, $R' = \infty$.

H4 is set so that P_N , P_R , P_P are all effectively equal to the DUMMY procedure defined as follows:
procedure DUMMY ; ;

H5 is set so that $P_{N'}$, $P_{R'}$, $P_{P'}$ are all effectively equal to DUMMY.

H6 is set so that $TAB = 1$.

H7 is set to terminate the program in case the data ends.

Step 2. (Layout)

The layout procedure is called: some of the variables H1, H2, H3, H4, H5, H6, H7 might be changed. T is set to false. T is a Boolean variable used to control the sequencing of data with respect to title formats; $T = \text{true}$ means a value has been requested of the procedure but has not yet been input.

Step 3. (Communication with list procedure)

The next format item of the format string is examined. (After the format string is exhausted, standard format is used until the end of the procedure. In particular, if the format string is ' ', standard format is used throughout.) If the next format item is a title format (requires no data item), step 4 is executed directly. If $T = \text{true}$ step 4 is executed. Otherwise, the list procedure is called using a procedure named INITEM as the actual parameter. Each subsequent return from INITEM causes the list procedure to be continued from the latest INITEM call. Since the scope of the identifier INITEM is local to INLIST, INITEM cannot be called directly. After the list procedure has been activated, it either terminates or calls INITEM. In the former case, the input process is complete; in the latter case, T is set to true and any assignments to hidden variables resulting from the list procedure cause adjustments to the variables H1, H2, H3, H4, H5, H6, H7 (which are local to INITEM); continue at step 4.

Step 4. (Alignment marks)

If the next format item includes an alignment mark at its left, it is removed from the format string and process A (a subroutine below) is executed for each /, process B for each ↑, and process C for each J. Overflow procedures might cause the format string to be changed. In this case, the new format string is not examined until step 6.

Step 5. (Get within margins)

Process G is executed to ensure proper page and line alignment.

Step 6. (Formatting for input)

The next item is taken from the format string. In unusual cases, the list procedure or an overflow procedure may have called the descriptive procedure `FORMAT`, thereby changing the format string. If so, the new format string is examined from the beginning; it is conceivable that the format items examined in steps 3, 4, 6 might be three different formats. At this point, the current format item is effectively removed from the format string and copied elsewhere so that the format string itself, possibly changed by further calls to `FORMAT`, is not examined until the next occurrence of step 3.

Alignment marks at the left of the format item are ignored. If the format item is not composed only of alignment marks and insertions, the value of `T` is examined. If `T = false`, undefined action takes place (a nontitle format has been substituted for a title format in an overflow procedure, which is not allowed). Otherwise `T` is set to `false`. If the format item is `A` or `N`, `s` is set to 1 and step 7 is executed; otherwise, the number of symbols, `s`, needed to represent the formatted item on the present medium, is determined from the format item.

Step 7. (Check for overflow)

If the present item uses `N` format, the character positions $\rho + 1, \rho + 2 \dots$ are examined until either a delimited number has been found (in which case ρ is advanced to the position following the delimiter, and step 9 is executed), or position $\min(R, P)$ has been reached with no sign, digit, decimal point or 10 encountered. In the latter case, step 8 is executed with $\rho = \min(R, P)$. If `N` format is not used, step 8 is executed if $\rho + s > \min(R, P)$, or step 9 if $\rho + s \leq \min(R, P)$.

Step 8. (Processing of overflow)

Process `H` ($\rho + s$) is performed and so is one of the following processes, depending on the current format item:

`N` format: Characters are input until a number followed by a delimiter is found, in which case step 9 is executed; or if position $\min(R, P)$ is reached, a partial number may have been examined. Step 8 is repeated until a number followed by a delimiter has been input.

`A` format: Characters are input as with `N` format until a basic symbol has been input. (This basic symbol may use several character positions on the input medium.)

Other: if $\rho + s \leq R$ and $\rho + s \leq P$, step 9 is executed; otherwise input $k = \min(R, P) - \rho$ characters, set $\rho = \min(R, P)$, decrease `s` by `k`, and repeat this step.

Step 9. (Finish the item)

If any format other than `N` and `A` is being used, `s` characters are input. The value of the item input is determined (steps 7 and 8 in the case of `N` and `A` format) using the format rules. This value is assigned to the actual parameter of `INITEM` unless a title format was specified. ρ is increased by `s`. Any alignment marks at the right of the format item now cause activation of process `A` for each `/`, process `B` for each `↑`, and process `C` for each `J`. Return to step 3.

Process `A` (`/` operation)

Page alignment is checked by process `F`, process `D` is executed, and procedure `PN` is called.

Process B (\uparrow operation)

If $\rho \geq 0$, process D is executed and procedure P_N is called. Then process E is executed and procedure P_N' is called.

Process C (J operation)

Page and line alignment are checked by process G. Then k is set to $((\rho - L + 1) \div \text{TAB} + 1) \times \text{TAB} + L - 1$ (the next tab setting for ρ), where TAB is the tab spacing for this channel. If $k \geq \min(R, P)$, process $H(k)$ is performed; otherwise character positions are skipped until $\rho = k$.

Process D (New line)

The input medium is skipped to the next line, ρ is set to 0, and ρ' is set to $\rho' + 1$.

Process E (New page)

If the input file is paged, records are skipped until a record is found which contains a new page mark. Otherwise, the input file is skipped to the next line. In either case ρ and ρ' are set to 0.

Process F (Page alignment)

If $\rho' + 1 < L'$ process D is executed until $\rho' = L' - 1$. If $\rho' + 1 > R'$, process E is executed, $P_{R'}$ is called, and process F is repeated. If $\rho' + 1 > P'$, process E is executed, $P_{P'}$ is called, and process F is repeated. Process F must terminate because $1 \leq L' \leq R'$ and $1 \leq L' \leq P$. If a value of $L' > P$ is chosen, L' is set to 1.

Process G (Page and line alignment)

Process F is executed. Then, if $\rho + 1 < L$ character positions are skipped until $\rho + 1 = L$. If $\rho + 1 > R$ or $\rho + 1 > P$ process $H(\rho + 1)$ is performed. This process must terminate because $1 \leq L \leq R$ and $1 \leq L \leq P$. If a value of $L > P$ is chosen, L is set to 1.

Process H (k) (line overflow)

Process D is performed. If $k > R$, P_R is called; otherwise, P_P is called. Then process G is performed to ensure page and line alignment. Upon return from any of the overflow procedures, any assignments to hidden variables that have been made by calls to descriptive procedures cause adjustments to the corresponding variables, local to INITEM, H1, H2, H3, H4, H5, H6, H7.

B.5.3 Skipping

Two procedures are available which achieve an effect similar to that of the "tab" key on a typewriter:

h skip (position, OVERFLOW)

v skip (position, OVERFLOW)

where position is an integer variable called by value, and OVERFLOW is the name of a procedure. These procedures are defined only if they are called within a list procedure during an in list or out list operation. For h skip, if $\rho < \text{position}$, set $\rho = \text{position}$; but if $\rho \geq \text{position}$, call the procedure OVERFLOW. For v skip, an analogous procedure is carried out: if $\rho' < \text{position}$, effectively execute process A of Section B.5.1 ($\text{position} - \rho'$) times; but if $\rho' \geq \text{position}$, call the procedure OVERFLOW.

B.5.3 Skipping

The procedures HSKIP and VSKIP have been replaced by the procedure TABULATION, described in section B.3.3.

B.5.4 Intermediate data storage

The procedure call

put (n, LIST)

where n is an integer parameter called by value and LIST is the name of a list procedure (Section B.4), takes the value specified by the list procedure and stores them, together with the identification number n. Anything previously stored with the same identification number is lost. The variables entering into the list do not lose their values.

The procedure call

get (n, LIST)

where n is an integer parameter called by value and LIST is the name of a list procedure, is used to retrieve the set of values which has previously been put away with identification number n. The items in LIST must be variables. The stored values are retrieved in the same order as they were placed, and they must be compatible with the type of the elements specified by LIST; transfer functions may be invoked to convert from real to integer type or vice versa. If fewer items are in LIST than are associated with n, only the first are retrieved; if LIST contains more items, the situation is undefined. The values associated with n in the external storage are not changed by get.

B.5.4 Intermediate Data Storage

The procedures GET and PUT are implemented as special cases of GETLIST and PUTLIST assuming the special channel number 0 and using an index derived from the integer parameter. This index is not available to the programmer.

B.6 Control Procedures

The procedure calls

out control (unit, x1,x2,x3,x4)

in control (unit, x1,x2,x3,x4)

may be used by the programmer to determine the values of normally "hidden" system parameters, in order to have finer control over input and output. Here unit is the number of an output or input device, and x1,x2,x3,x4 are variables. The action of these procedures is to set x1,x2,x3,x4 equal to the current values of ρ, P, ρ', P' , respectively, corresponding to the device specified.

B.6 Control Procedures

In the input/output system as described up to this point, the physical limits characteristic of the various devices (P, P'), the number of spaces (k) which serves as a number delimiter in standard format, and the current value of the character pointers (ρ, ρ') are effectively system parameters which are not directly accessible by the programmer. These quantities are accessible to, and in many cases modified by, several input and output procedures.

To obtain more explicit control over the input/output processes, the programmer can access these quantities through the procedure call

SYSPARAM (channel,function,quantity)

channel is an arithmetic expression, called by value, specifying the input/output device concerned.

function is an arithmetic expression, called by value, specifying the particular quantity to be accessed (defined below) and specifying whether that quantity is to be interrogated or changed.

quantity is an integer variable, called by name, that either represents the new value or is to be assigned the present value of the quantity, depending on function.

The following list defines the standard set of quantities accessible through **SYSPARAM** and the corresponding value of function.

For the external device associated with channel:

If function = 1, quantity := ρ

If function = 2, ρ := quantity[†]

If function = 3, quantity := ρ' (returns 0 for unpagged channels)

If function = 4, ρ' := quantity[†] (dummy function for unpagged channels)

If function = 5, quantity := P

If function = 6, P := quantity[‡]

If function = 7, quantity := P' (returns 0 for unpagged channels)

If function = 8, P' := quantity[‡] (dummy function for unpagged channels)

If function = 9, quantity := k (k must be ≥ 1)

If function = 10, k := quantity

ρ and ρ' are the character and line pointers.

P and P' are the physical limits of the device.

k is the number of blanks delimiting a standard number format.

[†]Since ρ and ρ' represent the actual physical positions on the external device, function = 2 or 4 generally causes some action to take place for that device. When ρ is set, if quantity $\geq \rho$, blanks are inserted until ρ = quantity. If quantity $< \rho$, a line advance operation is performed, ρ is set to 0, and blanks are inserted until ρ = quantity. When ρ' is set, if quantity $> \rho'$, line advance operations are performed until ρ' = quantity. If quantity $\leq \rho'$, the next page is skipped to, ρ' is set to 0, and line advance operations are performed until ρ' = quantity. The action for function = 2 or 4 depends on the last operation on the file. If it was read, characters are input; if it was write, characters or lines are output; if there was no previous operation or one that closed the file (such as re-wind), no action takes place.

[‡]These operations change the physical limits for the input/output device whenever possible (such as block length on magnetic tape). When the limits cannot be changed for the input/output device, these functions are equivalent to a dummy statement.

B.7 Other Procedures

Other procedures which apply to specific input or output devices may be defined at installations, (tape skip and rewind for controlling magnetic tapes, etc.). An installation may also define further descriptive procedures (thus introducing further hidden variables); for example, a procedure might be added to name a label to go to in case of an input error. Procedures for obtaining the current values of hidden variables might also be incorporated.

B.7 Other Procedures

The following additional procedures have been implemented. They are described fully in sections 3.2–3.5, and 3.10.

Primitive Procedures	{ CHLENGTH STRING ELEMENT
Input/Output Procedures for Direct Access Devices	{ GETLIST PUTLIST GETITEM PUTITEM GET PUT FETCHLIST STORELIST FETCHITEM STOREITEM
Input/Output Procedures for Binary Sequential Files	{ GETARRAY PUTARRAY
Control Procedures	{ ARTHOFLW PARITY EOF BAD DATA ERROR CHANERROR
Hardware Function Procedures	{ SKIPF SKIPB END FILE REWIND UNLOAD BACKSPACE

Miscellaneous Procedures

IOLTH
POSITION
DUMP
CLOCK
THRESHOLD
INRANGE
MOVE

Extended Core Storage/
Large Core Memory Procedures

READECS
WRITEECS

C. An Example

A simple example follows, which is to print the first 20 lines of Pascal's triangle in triangular form:

```

      1
    1  1
  1  2  1
1  3  3  1

```

These first 20 lines involve numbers which are at most five digits in magnitude. The output is to begin a new page, and it is to be double-spaced and preceded by the title "PASCALS TRIANGLE". We assume that unit number 3 is a line printer.

Two solutions of the problem are given, each of which uses slightly different portions of the input-output conventions.

begin integer N, K, printer;

integer array A[0:19];

procedure AK (ITEM); ITEM (A[K]);

procedure TRIANGLE; begin format ('6Z'); h lim (58 - 3 × N, 63 + 3 × N)
end;

printer : = 3;

output 0 (printer, ↑ 'PASCALS TRIANGLE' // ♪);

for N : = 0 step 1 until 19 do

begin A[N] : = 1;

for K := N-1- step - 1 until 1 do A[K] : = A[K - 1] + A[K];

for K : = 0 step 1 until N do out list (printer, TRIANGLE, AK);

output 0 (printer, ♪ // ♪)

end

end

```

begin integer N,K, printer;
  integer array A[0:19];
  procedure LINES;format 2('XB,X(6Z),//',57-3XN,N+1);
  procedure LIST(Q);for K : = 0 step 1 until N do Q(A[K]);

printer : = 3;

output 1 (printer, '↑20S//',PASCALS ▯ TRIANGLE);

for N : = 0 step 1 until 19 do
  begin A [N] : = 1;
    for K : = N - 1 step - 1 until 1 do A[K] : = A[K - 1] + A[K];
    out list (printer,LINES,LIST)
  end
end

```

D. Machine-dependent Portions

Since input-output processes must be machine-dependent to a certain extent, the portions of this proposal which are machine-dependent are summarized here.

1. The values of P and P' for the input and output devices.
2. The treatment of I, L, and R (unformatted) format.
3. The number of characters in standard output format.
4. The internal representation of alpha format.
5. The number of spaces, K, which will serve to delimit standard input format values.

REFERENCES

- Naur, P. (Ed.) Revised report on the algorithmic language ALGOL-60 Comm. ACM 6 (1963), 1-17.
- Extended ALGOL reference manual for the Burroughs B-5000. No. 5000-2102, Burroughs Corp., Detroit, 1963.
- SHARE ALGOL-60 translator manual. No. 1426, 1577, SHARE Distr. Agency. Oak Ridge ALGOL compiler for the Control Data 1604 computer. Oak Ridge Nat. Lab., Oak Ridge, Tenn.
- Duncan, F. G. Input and output for ALGOL-60 on KDF 9. Comp. J. 5 (1963), 341-344.
- Hoare, C. A. R. The Elliott ALGOL input/output system. Comp. J. 5 (1963), 345-348.
- McCracken, D. D. Guide to ALGOL programming. Wiley, New York, 1962. AED compiler. Electronic Systems lab., MIT, Cambridge, Mass.
- Ingerman, P. Z. A syntax-oriented compiler, etc. U of Penn., Moore School of Elect. Engineering, Philadelphia, Pa. 1963.
- Ingerman, P. A., and Merner, J. N. Revised revised ALGOL-60 report. Unpublished.

Perlis, A. J. A format language. Comm. ACM 7 (1964), 89-97.

Baumann, R. ALGOL-Manual der ALCOR-Gruppe, Elektron, Rechen, H. 5/6 (1961), H.2 (1962).

3.2 ADDITIONAL INPUT-OUTPUT PROCEDURES

3.2.1 PRIMITIVE PROCEDURES

An additional set of primitive procedures exists without user declaration, as follows:

CHLENGTH (string)
STRING ELEMENT (s1, i, s2, x)

CHLENGTH

CHLENGTH is an integer procedure with a string as a parameter. The value of CHLENGTH (string) is the number of characters of the open string (actual or formal) enclosed between the outermost string quotes.

CHLENGTH is defined as follows:

integer procedure CHLENGTH (string);
string string;
comment evaluate the number of character positions string would
require if output using S format;
<procedure body >

STRING ELEMENT

The procedure STRING ELEMENT is introduced to enable the scanning or interpretation of a given string (actual or formal) in a device independent manner. It assigns to the integer variable x an integer corresponding to the ith character of the string s1 as encoded by the string s2.

STRING ELEMENT can be defined as follows:

procedure STRING ELEMENT (s1, i, s2, x); value i; integer i, x;
string s1, s2;
comment select the ith symbol in s1, search string s2: if a
match is found assign to x the position number of the corresponding
symbol in string s2, if no match is found assign 0 to x;
<procedure body >

In effect, an OUTCHARACTER process is performed on the string s1 according to the integer variable i. An INCHARACTER process is then performed on the resulting character to the string s2, producing an integer value to be stored in the integer variable x, as follows:

```
REWIND (channel);
OUTCHARACTER (channel, s1, i);
REWIND (channel);
INCHARACTER (channel, s2, x);
```

3.2.2 INPUT/OUTPUT PROCEDURES FOR DIRECT ACCESS DEVICES

3.2.2.1 INDEXED LIST INPUT AND OUTPUT

```
GETLIST (channel, index, list)
PUTLIST (channel, index, list)
```

Channel, an integer parameter called by value, is the number associated with the input/output device. Index, an integer called by value, is used to identify the list of items on the input/output device. List is the name of a list procedure (see section B.4).

These two procedures form a pair. The effect of PUTLIST is to output to the output device the items presented by a call to the list procedure; the whole set is identified by index.

The effect of GETLIST is to input from the input device the items identified by index, and transfer the values to the items presented by a call to the list procedure.

Any item output by PUTLIST can be input to a compatible item by GETLIST where compatibility is defined as:

<u>item type for PUTLIST</u>	<u>item types for GETLIST</u>
real or integer array	real or integer array
Boolean array	Boolean array
real or integer value	real or integer variable
Boolean value	Boolean variable

If an attempt is made to use GETLIST with an incompatible item a fatal error message is issued.

In the case of arrays, the elements are taken in lexicographical order, without regard to any multi-dimensional structure. Any necessary type conversion takes place during GETLIST according to the standard rules. If an array on the input medium is larger than the GETLIST item, excess elements are ignored; if smaller, excess elements in the item are unchanged.

If, during the evaluation of an input/output item, an attempt is made to input or output to the same channel, the result is undefined. For example:

```
procedure A (x); procedure x;
begin GETITEM (3, 72, B); x (B);
end;
PUTLIST (3, 64, A)
```

In this example, the call to GETITEM is in error and a fatal error message is issued.

The items provided by the list procedure can be arrays or arithmetic or Boolean expressions. (See section B.4.1.1.)

An index is a nonnegative parameter used to identify the location of the items on the external device. If more than one list of items with the same index is output to an indexed file and an attempt is made to input a list with that index from that file, then the effect is undefined.

3.2.2.2 INDEXED ITEM INPUT AND OUTPUT

GETITEM (channel, index, e_1, e_2, \dots, e_n)

PUTITEM (channel, index, e_1, e_2, \dots, e_n)

These procedures may be defined in terms of GETLIST and PUTLIST as follows, where a is a list procedure and p is the corresponding item procedure:

procedure GETITEM (channel, index, e_1, e_2, \dots, e_n);

value channel, index; integer channel, index;

comment e_1, e_2, \dots, e_n may be any integer, real or Boolean variable, or any integer, real or Boolean array;

begin procedure $a(p)$; procedure p ;

begin $p(e_1); p(e_2); \dots; p(e_n)$ end;
GETLIST (channel, index, a)

end

procedure PUTITEM (channel, index, e_1, e_2, \dots, e_n)

value channel, index; integer channel, index;

comment e_1, e_2, \dots, e_n may be any of the above or any integer, real or Boolean expression;

begin procedure $a(p)$; procedure p ;

begin $p(e_1); p(e_2); \dots; p(e_n)$ end;
PUTLIST (channel, index, a)

end

3.2.2.3 INDEXED LIST INPUT AND OUTPUT ON STANDARD STORAGE MEDIA

GET (index, list)

PUT (index, list)

These procedures are the same as GETLIST and PUTLIST except that they always use channel 0, which specifies mass storage. Information output to this channel is lost on program termination. In order to use GET or PUT, the user must provide a channel statement defining channel 0 as an indexed sequential channel.

3.2.2.4 WORD-ADDRESSABLE LIST INPUT AND OUTPUT

FETCHLIST (channel, address, list)

STORELIST (channel, address, list)

Channel is an integer parameter called by value and is the number associated with the file on the external device. Address is an integer variable called by name; on entry it contains the word address within the file where items are to be stored or retrieved; on exit it contains the word address of the next item in the file. List is a list procedure giving the items to be stored or retrieved.

These two procedures form a pair. The effect of STORELIST is to output to the output device the items presented by a call to the list procedure, in successive words, starting from the item whose word address is given by address. The effect of FETCHLIST is to input from the input device values starting from the value whose word address is given by address and to assign them to the items presented by the list procedure.

The items presented by the list procedure can be of the following type:

1. an array — real, integer or Boolean
2. an arithmetic or Boolean expression (STORELIST only)

On the external device, word addresses are sequential starting from 1. Each value (real, integer or Boolean) occupies one address (one word). Values of different types cannot be mixed on the same channel.

3.2.2.5 WORD ADDRESSABLE ITEM INPUT AND OUTPUT

FETCHITEM (channel, address, e_1, e_2, \dots, e_n)

STOREITEM (channel, address, e_1, e_2, \dots, e_n)

These procedures can be defined in terms of FETCHLIST and STORELIST in a way entirely analogous to the definitions of GETITEM and PUTITEM in 3.2.2.2 above.

3.2.3 INPUT/OUTPUT PROCEDURES FOR BINARY SEQUENTIAL FILES

The procedures GETARRAY and PUTARRAY are provided for the retrieval and storage of arrays with binary sequential files.

GETARRAY (channel, destination)

PUTARRAY (channel, source)

Destination and source are the names of arrays.

GETARRAY reads one record of the same length as destination directly from the channel into destination. The record is not stored first in a format area and no regard is made for maximum record size. The record should contain the array arranged by rows (as defined in section B.5, Array Transmission).

PUTARRAY writes one record, equal in length to source, directly from source to the channel. The record is not stored first in a format area and no regard is made for maximum record size. The record reflects exactly the storage of the array in memory, by rows.

3.3 CONTROL PROCEDURES

ARTHOFLW (label)
PARITY (channel, label)
EOF (channel, label)
BADDATA (channel, label)

Each one of these procedures establishes a label to which control transfers in the event of an arithmetic error (overflow, underflow or division fault), irrecoverable parity error, end of partition, or mismatch of input data to the corresponding format. Each procedure can be called as many times as necessary to modify the label in the course of a program. PARITY, EOF, and BADDATA must be called once for each channel for which a label is to be established. If a procedure has not been called, or if the label is no longer accessible when the corresponding condition occurs, the object program terminates abnormally with a fatal error message.

If INLIST is in operation, a label may be established by NODATA (section B.3.4) instead of by EOF. During the execution of INLIST, any label established by NODATA takes precedence over an EOF label.

ERROR (key, label)

ERROR is an execution-time trapping procedure. Key is an integer and is the key of the particular execution error to be trapped. In the event of the keyed error occurring after a call to ERROR and being within its scope, the label is jumped to. The scope of a call of ERROR is defined as the smallest block or procedure body surrounding the call plus any procedures called from this scope.

The values of key for ERROR are:

- 0 : any error
- 1 : mode error
- 2 : array/switch error
- 3 : parameter mismatch
- 4 : standard function parameter errors
- 5 : stack overflow
- 6 : string element error

CHANERROR (channel, key, label)

CHANERROR is an execution time error trapping procedure for input/output entirely analogous to **ERROR**. Channel is an integer called by value.

The values of key for **CHANERROR** are:

- 0 : any error
- 1 : parity error
- 2 : end of partition
- 3 : bad data
- 4 : formatting errors
- 5 : illegal operation
- 6 : auxiliary procedure errors

3.4 HARDWARE FUNCTION PROCEDURES

A channel is input if last used for a read operation, output if last used for a write operation, and closed if not yet used or referenced by a closing procedure such as **ENDFILE**.

If any of the following procedures is called for an external device that cannot perform the operation, the procedure is treated as a dummy procedure; at the completion of the procedure, the channel is considered to be closed. On mass storage devices the procedures **REWIND** and **UNLOAD** position the external device to the beginning of information.

SKIPF (channel)

This procedure spaces forward past one end of partition on a coded or binary sequential file. It is treated as a dummy procedure on an output channel, on indexed and word addressable files, or if an attempt is made to skip beyond the end-of-information.

SKIPB (channel)

This procedure spaces backwards past one end-of-partition on a coded or binary sequential file. The channel is left positioned at the front of the data records preceding the end-of-partition passed. On an output channel, before the spacing occurs, any information in the format area is written and an end-of-partition is written and backspaced over. If the channel is associated with an indexed or word addressable file, or if an attempt to skip beyond the beginning-of-information is made, the procedure is treated as a dummy procedure.

ENDFILE (channel)

This procedure writes an end-of-partition on the external device. It is treated as a dummy procedure on an input channel. Before the end-of-partition is written, any information in the format area is written.

REWIND (channel)

This procedure positions the specified channel to the beginning-of-information. On output, before rewinding occurs, any information in the format area is written; an end-of-partition is written followed by an end-of-information.

UNLOAD (channel)

This procedure unloads the external device. On output, before unloading occurs, any information in the format area is written; and an end-of-partition is written and backspaced over.

BACKSPACE (channel)

This procedure backspaces past one record on a sequential file, that is, one unit record (print line or card image) produced by coded sequential input/output or one record produced by a PUTARRAY call. Backspace is only allowed on files of Record Manager record type F, S or W or blocked files with one record per block. Backspace can jump over an end-of-partition as well as data records.

If the channel was last used for a write operation, any information in the format area is written; and an end-of-partition is written and backspaced over. The channel is then backspaced over the intended record.

3.5 MISCELLANEOUS PROCEDURES AND FUNCTIONS

IOLTH (channel)

This integer procedure returns as its value the length of the last item list read from or written to the external device associated with the value of the integer parameter channel. This procedure can only be used with indexed, binary sequential or word addressable channels; use with other channels will give a result of zero.

The length of the last item list is 1 if the item list was a single value and is the number of elements if it was an array.

POSITION (element, item)

This integer procedure returns as its value the displacement of the element (which must be an array element) within the item (which must be an array). This displacement, when added to the word address of the item, gives the word address of the element.

DUMP (identifying integer, option)

This procedure may be used to obtain output of the local and formal variables in the currently active block or procedure body. The format is that of the object-time abnormal termination dump (chapter 13). The dump is entitled

DYNAMIC CALL TO DUMP NUMBER < identifying integer > AT LINE < line number >

Identifying integer is an integer type variable or constant

Option is an arithmetic expression taking on the same values as the post mortem dump options.

- 0 : no dump
- 1 : traceback of program execution
- 2 : reduced octal dump of local quantities and formals
- 3 : reduced symbolic dump (decimal) of local quantities and formals
- 4 : complete symbolic dump (decimal) of local quantities, formals and arrays.

Options 2 through 4 include the traceback.

The following convention has been added:

positive value of option : dump of the dynamic chain of execution

negative value of option: dump restricted to current block

CLOCK

This is a real procedure whose value at any instant is the elapsed CPU time in seconds at the control point at which the program is executing. The value is accurate to one millisecond.

THRESHOLD (function, destination)

This is an environmental enquiry procedure. Function is an integer specifying the quantity to be accessed and destination is a variable of suitable type to hold the quantity. The available values for function are:

<u>Function</u>	<u>Quantity to be accessed</u>
1	Largest integer absolute value
2	Largest real absolute value
3	Smallest real absolute value
4	Precision for unity

The largest integer N is defined as the representation in normalized floating point for which the following conditions are true:

$$\begin{aligned}N+1 &= N \\ N-1 &\neq N\end{aligned}$$

This number is $2 \uparrow 48$ which is 281 474 976 710 656

The largest real number distinguishable from machine infinity is

$$(2 \uparrow 48-1) * (2 \uparrow 1022) \text{ which is } 1.2_{10}^{322}.$$

The smallest number distinguishable from zero by the machine in ALGOL 4 is $(2 \uparrow 47) * (2 \uparrow -1022)$ which is approximately 1.6_{10}^{-294} . This is the smallest normalized number.

Precision for unity is defined to be the smallest real number that can be added to or subtracted from 1.0 in ALGOL 4 to produce a result distinguishable from 1.0. This number is $2 \uparrow -47$ which is approximately 7.1_{10}^{-15} .

INRANGE (param)

This Boolean function has one real value parameter that returns the value false if the parameter is infinite or indefinite, and true otherwise.

Specific representations exist for \pm infinity and for \pm indefinite. If such operands are used in arithmetic operations, a mode error will result.[†] In ALGOL, infinite operands are caused by overflow and indefinite operands are the result of dividing zero by zero. These operands can also be present in variables after storage space has been so preset and before the variable has been assigned a value by the program. Presetting of this kind could be either the result of the program loading operating or of object time stack space requests with the P option selected (chapter 8).

MOVE

procedure MOVE(A,I₁,I₂,...,I_k,B,J₁,J₂,...,J_p,N);

array A,B;

value I₁,I₂,...,I_k,J₁,J₂,...,J_p,N;

integer I₁,I₂,...,I_k,J₁,J₂,...,J_p,N;

The effect of a call to procedure MOVE is the transfer of N elements from array A starting at the element A[I₁,I₂,...,I_k] to array B starting at element B[J₁,J₂,...,J_p]. The procedure MOVE can perform the transfer from any type of memory to any type of memory (CM, SCM, ECS, LCM) whether the array is virtual or not.

The transfer is performed with the rightmost index varying most quickly (see chapter 14 for array organization in memory). If A is of type real and B is of type integer, during the transfer, the real-to-integer conversion is performed as discussed in 4.2.4, chapter 2.

The following checking is performed before the transfer takes place:

1. Parameter A must be an array.
2. The k following parameters must be of type integer or real and k must be equal to the number of dimensions of A.
3. The parameter B must be an array of a type compatible with A.
4. The p following parameters must be of type integer or real and p must be equal to the number of dimensions of B.
5. The parameter N must be of type integer or real and must be the last parameter of the procedure call.
6. Finally, starting addresses corresponding to A[I₁,I₂,...,I_k], B[J₁,J₂,...,J_p] and final addresses corresponding to the Nth linear element are further checked to see that they are within the global array field. When the array bound checking option is activated (see chapter 6), starting addresses A[I₁,I₂,...,I_k], B[J₁,J₂,...,J_p] are more accurately checked index by index.

3.6 INPUT/OUTPUT ERRORS

At object time, two types of errors not directly concerned with programming are detected: illegal input/output operation requests and invalid transmission of data (chapter 8).

[†]Under SCOPE 2 a mode error occurs during the faulty calculation rather than when the infinite or indefinite number is used in subsequent calculations.

3.6.1 ILLEGAL INPUT/OUTPUT OPERATIONS

The object program terminates abnormally with a diagnostic if:

An input (output) operation is requested on a channel associated with a device which cannot read (write), or on a device which is prevented by the operating system from reading (writing).

A read operation immediately follows a write operation or vice versa.

3.6.2 TRANSMISSION ERRORS

Transmission errors are first treated by standard recovery procedures. If an error persists, it is irrecoverable.

On an irrecoverable parity error, control transfers to the label established for the channel by the PARITY procedure. If the PARITY procedure was not called or if the established label is no longer accessible, the object program terminates abnormally with the diagnostic UNCHECKED PARITY.

3.7 END OF PARTITION

When an end of partition is encountered on a read on an input device, control transfers to the label established for the channel by the NODATA procedure (within INLIST only) or the EOF procedure. If neither procedure has been called and if no label established by either is accessible, the object program terminates abnormally with the message UNCHECKED EOF. During execution of the INLIST procedure, any label established by NODATA takes precedence over a label established by EOF.

3.8 EFFICIENT USE OF FORMATTED INPUT/OUTPUT

The simplest procedures for formatted input/output are INPUT and OUTPUT. If the optimizing mode has been selected, an attempt is made to analyze and simplify calls to these two procedures at compile time and if an actual call satisfies all the following conditions, a faster call will be made at execution time:

1. The list of items to be input or output contains only arrays and simple variables.
2. The format is valid and the format string is an actual string.
3. Each format item is a simple format item.
4. The format does not contain the alignment mark J.

Using the definitions of format items in chapter 3, section A, the following are defined:

<simple number format> ::= <number format without insertion sequences>

<simple format item> ::= <simple number format> | <standard format> |

<Boolean part> | <non format> | <S> | <A> | <insertion> | <alignment mark>

In general, a simple format item cannot contain associated or embedded alignment marks or insertion sequences, or J or X.

The following are examples of simple format items:

4DD	+3ZD.2D-ZD	M
N	P	R
L	A	I
5S	'A STRING'	F
6B	↑	/
H		

The following are valid format items but not simple format items:

'INTEGER' +ZD.2B 'FRACTION' 4D
 /F
 N/
 'ABC' 7S

The following declarations are assumed for the examples given below:

real a, b, c; integer i;

Boolean p;

array y [0 : 5];

SIN and COS are standard functions.

Examples

Calls that satisfy conditions 1 through 4

- (i) INPUT (60, ' ', a, y [i]);
- (ii) OUTPUT (2, '/', 'ANSWER', 3B, +3ZD, ' ', F', a,p);
- (iii) OUTPUT (2, '(/, 3(+2ZD,3B))', y);

Calls that do not satisfy all of the conditions 1 through 4:

- (iv) INPUT (60, 'J,N', a);
- (v) OUTPUT (7, ' ', a, SIN (a), COS (a) *2.0);
- (vi) OUTPUT (2, '/', 'ANSWER' 3B+3ZD, ' ', F', a,p);

The results of examples (ii) and (vi) are the same; however, example (ii) will require less time to execute.

3.9 EXTENDED CORE STORAGE, LARGE CORE MEMORY PROCEDURES

The procedures READECS and WRITEECS are provided for reading from or writing to ECS or LCM. They are included only for compatibility with previous versions of ALGOL.

It is recommended that the procedure MOVE be used instead (chapter 3, section 3.5).

READECS (address, array, label)

This procedure transfers the contents of successive locations from the user direct access area of ECS (or LCM) addressed by the integer expression, address, into the array in central memory. Control transfers to the label if an irrecoverable parity error occurs.

WRITEECS (address, array, label)

This procedure transfers the contents of the array to the user direct access area of ECS (or LCM) beginning at the first word address given by the value of the integer expression, address. Control transfers to the label if an irrecoverable parity error occurs.

Input to the compiler may be an ALGOL source program or an ALGOL source procedure. More than one source program or source procedure may be compiled with a single call to the compiler.

In the following definitions, the symbol eop indicates the delimiter 'EOP' which separates successive ALGOL programs.

4.1 SOURCE PROGRAM DEFINITION

The following definition for an ALGOL source program is based on the definition of an ALGOL program (section 4.1.1., chapter 2), plus the following definition of implicit outer block head.

4.1.1 SYNTAX

$\langle \text{implicit outer block head} \rangle ::= \langle \text{block head} \rangle;$
 $\langle \text{implicit outer block head list} \rangle ::= \langle \text{implicit outer block head} \rangle | \langle \text{implicit outer block head list} \rangle$
 $\langle \text{implicit outer block head} \rangle$
 $\langle \text{pre} \rangle ::= \langle \text{any sequence of symbols except } \underline{\text{begin}}, \underline{\text{code}}, \underline{\text{algol}}, \text{ or } \underline{\text{procedure}} \rangle | \langle \text{empty} \rangle$
 $\langle \text{post} \rangle ::= \langle \text{any sequence of symbols except } \underline{\text{eop}} \rangle | \langle \text{empty} \rangle$
 $\langle \text{source program} \rangle ::= \langle \text{pre} \rangle \langle \text{program} \rangle \langle \text{post} \rangle | \langle \text{pre} \rangle \langle \text{implicit outer block head list} \rangle$
 $\langle \text{program} \rangle \langle \text{post} \rangle$
 $\langle \text{source program list} \rangle ::= \langle \text{source program} \rangle | \langle \text{source program list} \rangle \underline{\text{eop}} \langle \text{source program} \rangle$

4.1.2 SEMANTICS

A source program must contain declarations of all variables referenced in it. It must contain declarations for all procedures (except standard) it calls, including those that are compiled separately from the main program as ALGOL source procedures (section 5.4.6, chapter 2). However, the facility exists for adding implicit outer blocks to a program at compile time. This allows the user to reference identifiers in the program that are not explicitly declared therein but which will be present in the added outer blocks. A source library of such outer blocks can be maintained and included at compile time via control statement options. Any number of such blocks may be added provided that the nested block level does not exceed the imposed limit.

Implicit outer blocks cause nesting of the source program. To ensure syntax correctness, the compiler automatically supplies as many occurrences of end as necessary and issues a warning message.

Compilation of an ALGOL source program (generation of object code) starts with the ALGOL symbol begin in the source deck and terminates with the end symbol that causes the number of begin and end symbols to be equal, or with the eop, whichever occurs first; however, a diagnostic is issued if the number of occurrences of begin is not equal to the number of occurrences of end.

Any information in the source deck prior to the first begin or between the final end and the eop is treated as a commentary, listed as part of the source listing and included in the line count.

A program name is generated from the characters in columns one through seven of the first source line, provided the character in column 1 is alphabetic. This name is terminated with the seventh character or by the first character encountered that is not a letter or digit. If the character in column 1 is not alphabetic, the name generated is XXALGOL. The generated name is assigned to the binary main program created from the source program (chapter 5) and is listed on the page headings of the source listing.

4.2 SOURCE PROCEDURE DEFINITION

The following definition of an ALGOL source procedure is based on the definition of a procedure declaration in the ALGOL-60 Revised Report (chapter 2, section 5.4.1).

4.2.1 SYNTAX

`<pre> ::= <empty> | <any sequence of symbols except begin, code, algol, or procedure>`
`<mid> ::= <empty> | <any sequence of symbols except procedure, real, integer, or Boolean>`
`<post> ::= <empty> | <any sequence of symbols except eop>`
`<code number> ::= <sequence of one to five digits>`
`<external identifier> ::= <identifier with up to 7 digits and letters, the first a letter>`
`<code identifier> ::= <code number> | <external identifier> | <empty>`
`<code head> ::= <pre> code <code identifier>; <mid> | <pre> algol <code identifier>; <mid>`
`<source procedure> ::= <code head>; <procedure declaration>; <post>`
`<source procedure list> ::= <source procedure> | <source procedure list> eop <source procedure>`

4.2.2 SEMANTICS

A source procedure must contain declarations for all variables referenced in it. It must contain declarations for all procedures (except standard) it calls, including procedures that are compiled separately as ALGOL source procedures (section 5.4.6, chapter 2).

A source procedure may employ the same language features as a procedure declared in a source program.

Compilation of an ALGOL source procedure is initiated by the ALGOL symbol code or algol. This symbol is followed by the code identifier which is either a number in the range 0 through 99999, an external identifier of seven or fewer letters or numbers beginning with a letter, or empty, and then by a semi-colon. In the case where it is empty, the name declared in the procedure heading is taken to be the external identifier of the ALGOL source procedure. The same code number or external identifier is included in the body of the declaration for this procedure in the source program or source procedure referencing it. (section 5.4.6, chapter 2).

Compilation of an ALGOL source procedure starts with the symbol procedure which may be preceded by one of the type declarators real, integer, or Boolean.

If the procedure symbol is encountered before the code or algol symbol, compilation of the procedure starts normally, but an error message is issued and a code number of 00000 is supplied.

Compilation of an ALGOL source procedure ends at the normal end of the procedure declaration. If the body of the procedure is a single statement, the end is at the semicolon terminating that statement. If the body is a compound statement or block, the end is at the semicolon following the balance of begin and end symbols. If the eop occurs before the single statement is complete or before begin and end symbols balance, a diagnostic is issued.

The name generated for the procedure is the external identifier or CXXXXX, where XXXXX is the code number, when the symbol code is used. If five digits are not specified the number is zero-filled on the left. For example, 20 becomes 00020. Any error in the specification of the code number or external identifier results in the number 00000. When the code identifier is empty, the name of the procedure declared in the code body is assumed to be the external identifier. The generated name is assigned to the binary subprogram output from the source procedure and is also printed on the page headings of the source listing.

A source procedure cannot contain overlays.

4.3 SOURCE INPUT RESTRICTIONS

A single source program or single source procedure, or any combination of these, may be compiled with one call of the compiler.

The object program resulting from the compilation of a single source program, with no special binary subprogram input, is always executable, provided there are no compilation errors.

4.4 LANGUAGE CONVENTIONS

The input file contains the character representations for the ALGOL symbols shown in appendix C.

A blank character has no effect on the compilation process, except in strings (chapter 2). Blanks may be freely used elsewhere to facilitate reading. For example, MEAN UPPER BOUND, MEAN UPPERBOUND, and MEANUPPERBOUND are treated as being identical (the same name). Similarly, blanks may be included in the character representation of the ALGOL symbols. The ALGOL symbol real may be coded as 'R E A L' instead of the normal 'REAL'.

4.5 SOURCE LINE CONVENTIONS

A source line is equivalent to a card (in the case of source input originating through the card reader) or to a card image (in the case of source input originating from a terminal or from some other device). The broadest definition of a source line is a single Record Manager Z type (zero byte terminated) record. The maximum length of a source line is 126 characters.

An installation option specifies the number, N, of leftmost characters of a source line to be interpreted as significant by the compiler. The default value for N is 72. Characters between N + 1 and 126 are listed by the compiler but are not used as part of the source program or procedure. Any language construction can be continued across source lines; only the first N characters of each line are used. At compile time, a line number is assigned to each source line, beginning at 1. This line number is listed along with the source line in the output listing and is used in error messages.

The K option of the ALGOL control statement (chapter 6) can be used to override the installation setting of N. If K is not used, the installation setting remains in effect.

4.6 SOURCE DECK

A source deck consists of the source lines constituting one source program or one source procedure. The source decks to be compiled by a single call to the compiler occur successively on the source input file (the file specified by the I option of the ALGOL control statement). Any number of source decks in any order (within the restrictions described above) can occur as one section on the source input file. Each ALGOL control statement processes all the source decks in one section of the source input file. (In the case of files originating through the card reader, a section is a group of cards delimited by 7/8/9 cards.) If more than one source deck is submitted to the compiler, each source deck must be separated from the following by the delimiter eop. The characters 'EOP' can be placed anywhere in the source line; the following source deck, however, must begin on a new source line. The last source deck must not be followed by an eop, but rather by an end-of-section (7/8/9 card) or an end-of-information (6/7/8/9 card).

5.1 BINARY OUTPUT

The B option of the ALGOL control statement determines the disposition of binary object code produced as a result of an ALGOL compilation. If B is set to 0, compilation proceeds normally except that no binary output is produced. Binary generation is also inhibited if fatal errors are encountered during compilation. In all other cases, relocatable binary object code is generated and written to the file specified by the B option (or to the file LGO if the B option is omitted). The maximum size of the binary output generated from a single source program or source procedure is 131,072 words.

The format of the relocatable object code is suitable for loading and execution by the system loader. If the object program, together with its data requirements, is too large to fit into available central memory, field length requirements can perhaps be reduced if the program is rewritten using the virtual or overlay delimiter, and re-compiled using the S or C control statement option (chapters 6 and 10).

The binary object code generated for source programs is in the form of a main program; the binary object code generated for a source procedure is in the form of a subprogram. To be successfully loaded and executed, the binary output file should contain only one main program but can contain any number of subprograms. The subprograms can be produced in any way (by ALGOL or FORTRAN Extended compilation, or by COMPASS assembly), provided that the linkage conventions defined in chapter 2, section 5.4.6 are observed.

5.2 ASSEMBLY LANGUAGE OBJECT CODE

The compiler generates the object code directly into binary form, with no intermediate assembly language form. If the A option of the ALGOL control statement is selected, however, the compiler produces and lists a coded version of the binary output in COMPASS format. If the P option of the ALGOL control statement is selected, the compiler writes the COMPASS version of the object code in punchable form to the file specified. The listing has the same format as a COMPASS listing, with each COMPASS instruction appearing on one line. The punch form results in a legal COMPASS assembly deck, with one COMPASS instruction punched in the proper positions on each card.

5.3 SOURCE LISTING

The user may request a printed listing of any source program or source procedure compiled. Each line in the listing corresponds to one source line (as defined in chapter 4). The lines appear in the same order as in the source deck, right shifted for readability.

Each source line in a source deck is assigned a line number by the compiler, beginning with 1. Every line of the listing contains the number of the source line.

Diagnostics generated during compilation are printed following the source listing. Each contains a summary of the error condition and the approximate source line number on which the error was detected.

Diagnostics are printed even if the source listing is suppressed. Advisory diagnostic can be suppressed by means of the N parameter on the ALGOL control card (section 6.1.2). Chapter 15 contains a description of the diagnostics.

The ALGOL compiler is called from the system library by the ALGOL control statement. The files to be used for input and output and the type of output to be produced, as well as other options, are specified in any of the following formats:

ALGOL (p₁, p₂, . . . , p_n) comments

ALGOL, p₁, p₂, . . . , p_n. comments

ALGOL. comments

Example:

ALGOL (C=2, R)

The optional parameters, p₁, . . . , p_n, may be in any order. All parameters must be separated by commas. If no parameters are specified, ALGOL is followed by a period or right parenthesis. If a parameter list is specified, it must conform to the control statement syntax as defined in the appropriate operating system reference manuals, with the added restriction that the only valid delimiter between parameters is the comma. Comments can follow the right parenthesis or period; they are ignored by the compiler, but are listed in the dayfile.

Default values, which are set when the system is installed, are used for omitted parameters. The defaults initially released with the system are listed below; any of these can be changed by an installation.

6.1 CONTROL CARD PARAMETERS

6.1.1 SYNTAX

<external identifier> ::= <identifier with 1 to 7 letters and digits, the first a letter>

<value> ::= <digit> | <digit> <digit> | <external identifier>

<value list> ::= <value> | <value list> / <value>

<keyword> ::= I | U | L | A | R | B | P | Ø | C | E | N | D | F | S | K | X | Q | G

<parameter> ::= <keyword> | <keyword> = <value list>

<parameter list> ::= <parameter> | <parameter list> , <parameter>

<control statement> ::= ALGOL, <parameter list> . | ALGOL (<parameter list>) | ALGOL.

The number used for <value> in a <value list> is limited according to the keyword.

6.1.2 SEMANTICS

I : source input is on standard input file (INPUT)

$I = 0$: no source input.

L List control (Default: **L = OUTPUT**)

L = fn : list source program, fatal diagnostics on file fn.

L = 0 : list only fatal diagnostics on standard output file (OUTPUT)

R	Cross reference map	(Default : 0)
---	---------------------	---------------

R = 0 : no cross reference map.

A Assembly listing (Default : A = 0)

A = 0 : no assembly language listing.

N	Advisory diagnostics.	(Default : N)
---	-----------------------	---------------

N = 0 : list of advisory diagnostics is suppressed; only diagnostics fatal to code generation are listed.

0	Optimization of generated code	(Default : 0 = 0)
---	--------------------------------	-------------------

O = 0 : compile program in the fast compile mode. This generated code is produced in the most general fashion without regard to special situations which can give rise to more efficient code.

O = 1 : perform linguistic optimization by optimizing procedure calling. Eliminates certain redundant compilations. For a further explanation of optimization, see chapter 12.

O = 2 : perform **O = 1** optimizations and also subscript and for-statement optimization.

O : same as **O** = 0

U	User implicit block head input	(Default : U = 0)
---	--------------------------------	-------------------

U : user implicit outer block head input supplementary to file specified by I option is on file
COMPILE.

U = fn : precede source program by the implicit outer block head list on file fn. Block heads precede the source program in their sequential order and must be in source form.

U = 0 : no file for implicit outer blocks.

For further information on implicit outer block heads, see chapter 4.

C	Comments interpretation for special delimiters	(Default : C = 0)
---	--	-------------------

C : same as C=0

C = 0 : no comments interpretation.

C = 1 : debugging directives which are present in comments are detected by the compiler and cause debugging code to be inserted into the object program.

C = 2 : overlay directives which are present in comments are detected by the compiler and cause overlay directives in correct loader format to be inserted into the object program.

C = 3 : array bound checking directives which are present in comments are detected by the compiler. The `checkon` directive causes array bound checking at execution time for each index of an array. `Checkon` and `checkoff` directives are followed by a list of array identifiers. All the arrays in the list of a `checkon` directive are checked until a `checkoff` directive for those arrays is encountered.

This option requires the compiler to search for and interpret additional delimiters in comments.

The delimiters are:

- a) debugging directives trace, snap, snapoff (chapter 9)
- b) overlay directive overlay (chapter 10)
- c) array bound checking directives checkon and checkoff

Multiple selection for the C option can be performed by separating each value by a slash. For example C = 3/2/1 is acceptable.

The virtual array directive is always detected by the compiler and is not dependent on the C option.

S Array storage allocation (Default : S = 0)

S : same as $S = 0$

S = 0 : all arrays are allocated to CM (or SCM).

S = 1 : virtual arrays are allocated to ECS (or LCM).

S = 2 : all arrays are allocated to LCM. This option applies only for programs to be executed on a CYBER 70 Model 76.

See chapter 11 for an explanation of ECS/LCM usage; S must not be used with O.

- P** Punch assembly language (Default : P = 0)
- P : punch assembly language form of the object code in standard assembly language card format on standard punch file (PUNCH).
- P = fn : punch assembly language on file fn.
- P = 0 : suppress assembly language punching.
- P should not be used in overlay mode.
-
- E** Exit parameter (Default : E = 0)
- E : abort the job to an EXIT control statement after compilation if a fatal error has occurred.
- E = 0 : suppress abort in case of fatal error.
-
- B** Object program in standard relocatable binary form (Default : B = LGO)
- B : output object program to file LGO.
- B = fn : output object program to file fn.
- B = 0 : no binary object program.
-
- D** Dumpfile assignment (Default : D = 0)
- D : create the symbol file on standard dump file DUMPFIL.
- D = fn : create the symbol file on file fn.
- D = 0 : suppress the symbol file.
- To have a symbolic dump at execution time (option D \geq 3 in chapter 8), the symbolic file must be created at compile time.
-
- F** Fatal error termination (Default : F = 0)
- F : if a fatal error is found in the first pass (ALG1), terminate the compilation at the end of this pass.
- F = 0 : continue until the normal end of compilation.
-
- K** Input record size (Default : K = 72)
- K : 72 significant characters.
- K = n : n is the number of significant characters to be interpreted by the compiler on the source line.
- The maximum number of characters that can be interpreted is K = 126.
-
- X** Real - integer correspondence between formal and actual parameters (Default : X)
- X = 0 : forbid any real-integer (or integer-real) correspondence between formal and actual parameters.
- X : allow real-integer (or integer-real) correspondence between formal and actual parameters, and in the case real to integer perform the conversion.
- Selection of this option significantly degrades the performance of the program.

Input/output procedures specify a channel on which the operation is to be performed; each channel is referenced by an identification number called a channel number (chapter 3, section B.1.1). Each channel is associated with a set of characteristics, some of which are defined with channel statements.

Channel statements appear as the first section of the channel statement input file (option C in chapter 8); if C = 0 is selected, no channel statements are read and only standard channels 60 and 61 are defined; they are interpreted by the controlling routine before the object program is entered. The two types of channel statement are channel define and channel equate; all must contain the characters CHANNEL, in columns 1 through 8.

The relationship between the structure of a file created by the input/output statements of a program and its physical representation is defined by the channel statement. The restrictions imposed by the operating system must be considered in creating a channel statement.

7.1 CHANNEL DEFINE STATEMENT

The channel define statement describes the characteristics to be associated with one channel number.

CHANNEL, cn=file name, p₁, p₂, . . . , p_n

The eight characters CHANNEL, appear in columns 1 through 8 followed by a list of parameters; spaces are not allowed in the parameters and cause termination of the statement.

Each parameter p_i, defined below, describes a different characteristic. Parameters are separated by commas. The last parameter has no delimiter, but the information for one channel must be contained on a single card image or terminal line. Only the cn=file name parameter is required; the others are optional and may be specified in any order.

cn channel number, unsigned integer, maximum 14 decimal digits

file name logical file name of 1-7 letters and digits beginning with a letter

The parameter defining the type of file can be:

C Coded sequential

B Binary sequential

I Indexed[†]

W Word addressable

Only one of these can appear. The default is C.

If the file is specified as binary sequential, no other parameters can be used.

[†]SCOPE 2 does not support indexed files.

7.1.1 SEQUENTIAL FILE PARAMETERS

Any of the following parameters may be included when the C file type option or no file type option has been specified.

- Pr r indicates maximum line width; when omitted P136 is assumed.
- PPs s indicates maximum number of lines per page. If PP0 is specified or if the parameter is omitted, no paging operations are performed. If the user defines page width or page length beyond the capabilities of the corresponding external device, data may be lost.
- Kb b determines the number of consecutive blanks that serve as a delimiter for a number read or written in standard format. Omission of this parameter is equivalent to K2. The number specified must be in the range $1 \leq b \leq r$.

7.1.2 INDEXED FILE AND WORD ADDRESSABLE FILE PARAMETERS

The following parameter may be included when the I or W file type option has been specified.

- Ls s indicates the size of the working storage area in words. The default value is 16 for indexed and 64 for word addressable channels. When each invocation of a standard input/output routine transfers a large amount of data, the number of input/output calls is reduced.

7.2 CHANNEL EQUATE STATEMENT

Channel equate statements permit the user to reference the same channel with more than one channel number:

$$\text{CHANNEL, } cn_1 = cn_2$$

cn_1 and cn_2 are unsigned integers with a maximum of 14 decimal digits each.

Either cn_2 , or a number to which cn_2 is linked by other channel equate statements, must appear on a previous channel define statement. The channel defined on that statement can be referenced by the number cn_1 as well as cn_2 . Any number of channel numbers may be equated in this way with the same channel.

7.3 DUPLICATION OF CHANNEL NUMBERS

Although a channel can be associated with more than one channel number, a channel number must refer to only one channel. Therefore, the same channel number must not appear in more than one channel define statement in a set. Similarly, a channel number which appears on a channel define statement cannot be included on the left-hand side of a channel equate statement, since this is equivalent to associating that number with more than one channel.

7.4 DUPLICATION OF FILE NAMES

The following rule applies to both user-defined channels and those automatically supplied by ALGOL.

A file name can appear on any number of channel define statements. The Run-Time System allocates separate buffers for each channel. Consequently, input/output operations occurring to or from the physical storage media take place in the order of buffer emptying or filling, which is not necessarily related to the order of logical input/output requests called for by the user program. Thus, input or output data processed by the program will not always be in the same sequence as that data appears on the external device. This problem can be avoided by not duplicating file names.

7.5 STANDARD ALGOL CHANNEL STATEMENTS

In the noninteractive mode of execution (execution time option Q = 0) two channel statements with standard channel numbers and characteristics are automatically supplied by ALGOL for the standard system input and output devices, as follows:

CHANNEL, 60 = INPUT, P80

CHANNEL, 61 = OUTPUT, P136, PP60

The two standard files may be referenced by the channel numbers 60 and 61 and do not require channel statements; however, the statements are listed as part of the channel statement listing as if they were specified by the user.

Under interactive mode (see chapter 16), the standard input and output device is the terminal. The user must supply any input data on file INALG (channel 60) and obtains his output on file OUTALG (channel 61).

The following channel statements are supplied by ALGOL:

CHANNEL, -2 = HISTORY, P136, PP60

CHANNEL, -1 = OUTPUT, P50, PP130000

CHANNEL, 60 = INALG, P80

CHANNEL, 61 = OUTALG, P136, PP60

Channels -1 and -2 are used only internally by the interactive monitor, and user access to them is forbidden.

7.6 TYPICAL CHANNEL STATEMENTS

Some typical channel statements are:

CHANNEL, 35 = NUCLEAR, P120

CHANNEL, 47 = UNCLEAR, P400

CHANNEL, 29 = 35

CHANNEL, 4 = DISK, W, L200

Run-time options control the execution of an ALGOL program. Options are specified by parameters on the program call control statement, the LGO control statement, or the EXECUTE control statement that initiates execution of the ALGOL program, in one of the following ways:

```
LGO(<parameter list>)
LGO,<parameter list>.
EXECUTE(entry point,<parameter list>)
EXECUTE,entry point,<parameter list>.
lfn(<parameter list>)
lfn,<parameter list>.
```

Default values, which are set when the system is installed, are used for omitted parameters. The defaults initially released with the system are listed below; any of these can be changed by an installation.

8.1 OPTIONS

8.1.1 SYNTAX

The parameter list must conform to the following syntax:

```
<external identifier>::=<1 to 7 letters and digits beginning with a letter>
<value>::=<digit> | <value><digit>
<value list>::=<value> | <value>/<external identifier>
<keyword>::= S | D | C | P | T | Q
<parameter>::=<keyword> | <keyword> = <value list>
<parameter list>::=<parameter> | <parameter list> , <parameter>
```

8.1.2 SEMANTICS

(Default : S = 0)

S Stack statistics

S = 0 : no stack statistics are output

S or S = 1 : output of stack statistics at job termination

When this option is selected the stack size (SCM/CM and LCM/ECS) is monitored during program execution. The amount of unused core in each stack is recorded on the file OUTPUT at termination. When the G option has been selected at compile time, the number of swaps performed during execution of the program is also given.

Stack size monitoring degrades program performance.

D Abnormal termination dump format (Default : D = 0)

D = 0 : no dump

D or D = 1 : traceback of program execution

D = 2 : traceback and reduced octal dump of local quantities and formal parameters

D = 3 : traceback and reduced symbolic dump (decimal) of local quantities and formal parameters

D = 4 : traceback and complete symbolic dump (decimal) of local quantities, formal parameters and arrays

The dump file name is specified by adding /lfn to the option:

D=i/lfn

The file must be the same as the dumpfile created with the D option at compile time. The default file name is DUMPFIL both at compile time and at run time.

For a further discussion of abnormal termination dumps see chapter 13.

C Channel statement input file (Default : C = 0)

C = 0 : no channel statement input file

C : channel statements are read from the file INPUT

C = lfn : channel statements are read from the file lfn

Channel statements are discussed in chapter 7.

P Presetting of arrays (Default : P = 0)

P : same as P = 0

P = 0 : no array presetting takes place

P = Z : arrays are preset to zero

P = U : arrays are preset to floating point undefined

If this option is selected, arrays are preset as specified.

T Debugging and dump output limit (Default: T = 1000)

T : same as T = 0

T = 0 : no debugging or dump output

T = n : only n messages are output

If this option is selected, debugging and dump output are suppressed after n messages have been output. Program execution continues.

Q Interactive debugging (Default : Q = 0)

Q = 0 : execute the program in noninteractive mode

Q : execute the program in interactive mode

Q = lfn : execute the program in interactive mode with the file lfn as the interactive file

A program compiled under the Q option must be executed in interactive mode, and a program not compiled under the Q option cannot be executed in interactive mode.

The same interactive file must be specified at compile time and at run time. The default file name is QFILE at both times.

See chapter 16 for a further discussion of the ALGOL Interactive Debugging Aids (AIDA).

8.2 EXAMPLE

LGO(C, P=Z, D=3/MYDUMP)

The program to be executed is on the file LGO. The channel statements are to be read from the file INPUT, and any arrays are to be preset to zero. If the program terminates abnormally, a traceback and reduced symbolic dump are to be produced. The information needed to produce these dumps can be found on the file MYDUMP.

An ALGOL program can be debugged within the context of the ALGOL language — without resorting to core dumps or object code listings in assembly language. A compile time option and debugging directives which are inserted into the source program are used. Debugging directives need not be removed from a program when debugging is not wanted since they have no effect unless the debugging option is specified.

9.1 GENERAL DESCRIPTION

When the C=3 option is selected at compile time, debugging directives in the source program cause code for monitoring program execution to be inserted into the object program. If the C=3 option is not selected, the debugging directives are considered as comments.

Monitoring consists of listing informative messages on the OUTPUT file when a monitored identifier is referenced. This output may be limited or suppressed by selecting the T option at run time (see chapter 8).

The C=3 and O=1 or O=2 options may not be used at the same time. If O=0 is not the default option, it should be specified whenever the C=3 option is specified; otherwise the results are unpredictable. The program should be recompiled after it has been debugged if a higher level of optimization is desired.

9.2 DEBUGGING DIRECTIVES

There are three debugging directives: trace, snap, and snapoff.

9.2.1 SYNTAX AND SEMANTICS

`< identifier list > ::= < identifier > | < identifier > , < identifier list >`

`< directive > ::= trace | snap | snapoff`

`< debugging directive > ::= < directive > < identifier list >`

At least one identifier must be specified for a debugging directive; the order of identifiers in the list is irrelevant.

A debugging directive is part of a comment sequence. Each comment sequence can contain only one debugging directive, and it must appear last, just before the terminating semicolon. Other text may precede the debugging directive. Directives are not recognized in comments following end even if the C=3 option is selected.

Examples:

```
comment***snap A,B;  
  
begin comment first of all trace L1, Failure;  
  
comment snapoff A;
```

9.2.2 TRACE DIRECTIVE

The trace directive monitors the flow of control of an executing program. It can appear anywhere in a block, but identifiers in the list must be declared in that block as either labels or procedures. The trace directive does not monitor any identifier declared in an outer block or a disjoint block or after subsequent re-declaration in an inner block.

Monitoring within any specific area of the program cannot be suppressed. Any transfer, reference, or call to an identifier specified in a trace directive while its associated declaration is within the current scope is monitored throughout program execution, including references by means of actual-formal correspondence. Monitoring does not take place, however, for code that occurs before the trace directive. For example, a go to statement preceding a trace directive is not detected.

9.2.3 SNAP DIRECTIVE

The snap directive monitors changes in the value of specified program variables during program execution. It can appear anywhere in a block. Unlike the trace directive, the identifiers in the list need not be declared in the same block as the snap directive; however, they must be declared as simple variables or as arrays in the block where the snap directive occurs or in an outer block.

An identifier specified in a snap directive is monitored only in statements which occur between the snap directive and the end of the block in which the identifier is declared. Since the snap directive is interpreted at compile time, the definition of the range of a snap directive is textual rather than dynamic.

9.2.4 SNAPOFF DIRECTIVE

The snapoff directive turns off monitoring initiated by a snap directive. It has no effect on trace directives. Monitoring is turned off for the identifiers specified in the identifier list of the snapoff directive. Monitoring can be turned on again by a subsequent snap directive.

The snapoff directive can occur in any part of the source program. Since the snapoff directive is interpreted at compile time, the definition of the range of a snapoff directive is textual rather than dynamic.

9.3 MONITORING MESSAGES

When a monitored identifier is referenced during execution, one of the following messages is written to the OUTPUT file. Each message names the identifier as well as the line number of the reference. The line numbers in the messages agree with those in the source program listing generated by the compiler.

****LINE** linenum identifier AS PARAMETER

A monitored simple variable is an actual parameter of a procedure being called.

****LINE** linenum identifier [*] AS PARAMETER

A monitored array or a subscripted variable is an actual parameter of a procedure being called.

****LINE** linenum identifier : = value

The value of a monitored simple variable is changed.

****LINE** linenum identifier [*] : = value

The value of a monitored array or a subscripted variable is changed.

****LINE** linenum GOTO identifier FROM LINE line number

Control is transferred to a monitored label.

****LINE** linenum CALL TO identifier FROM LINE line number

A monitored procedure is called.

****LINE** linenum EXIT FROM identifier

An exit is made from a monitored procedure.

9.4 MONITORING OF SIMPLE VARIABLES

Simple variables of the types integer, real, or Boolean can be monitored. A message is generated for an identifier referenced in a snap directive when any of the following occur:

The variable occurs on the left side of an assignment statement (its value is changed)

The variable is the controlled variable in a for-statement

The variable is an actual parameter in a procedure statement and that procedure is called

The use of the value of the variable in an expression is not monitored.

When the monitored variable is type integer or real, the value is output in standard decimal format. The output representation for a Boolean variable is either true or false.

Example:

The following lines of source code:

```

114 comment snap M, N;
115 M:=2*unstack(N)+1;

```

produce the following monitoring messages:

```
**LINE 115 M:=1.1000000000000000# +001
```

****LINE 115 N AS PARAMETER**

If a variable is an actual parameter to a procedure or function and call by value is not specified, the value of the variable changes whenever the value of the corresponding formal parameter changes. No code is generated to monitor these changes as the procedure body is not always available when the calling program is compiled. Instead, a message is generated when a variable being monitored is used as an actual parameter to a procedure or function. This message should be taken as a warning that the value of the variable may have changed even though no other monitoring message is generated.

Example:

The following lines of source code:

```

101 comment check the determinant, snap DET;
102 invert(array, DET);
103 comment DET is the determinant of array, snapoff DET;

```

produce the following monitoring message:

****LINE 102 DET AS PARAMETER**

9.5 MONITORING OF ARRAYS

A message is generated for an array element referenced in a snap directive when any of the following occur:

The array element occurs as a left side of an assignment statement (its value is changed)

The array element is an actual parameter in a procedure statement and that procedure is called

Use of the value of the array element in an expression is not monitored.

Monitoring messages for array elements do not indicate the value of the subscript. The subscript appears as an asterisk in the message.

Examples:

The following lines of source code:

```
126 comment snap A;  
127 A[J, 3*J-2]:=2**5;
```

would produce the following monitoring message:

```
**LINE 127 A[ * ] :=3.2000000000000000≠+001
```

The following lines of source code:

```
52 comment snap A,B,C;  
53 if I=0 then p(A, B [ J ]+1, C [ J ] );
```

where A, B, and C are array identifiers, produce the following monitoring messages:

```
**LINE 53 A [ * ] AS PARAMETER
```

```
**LINE 53 C [ * ] AS PARAMETER
```

The reference to B[J] is not monitored because its value is used in an expression.

9.6 MONITORING OF LABELS

A message is generated whenever control is transferred, other than by sequential execution, to a label specified in a trace directive. Transfer is by a go to statement, either unconditional or as a result of evaluation of a designational expression. The transfer is monitored also if the label is obtained as a switch element or by use of a corresponding formal parameter label identifier.

Example:

```
**LINE 231 GO TO INFEAS FROM LINE 120
```

9.7 MONITORING OF PROCEDURES

A message is generated whenever a procedure or function specified in a trace directive is called. The message includes the line number where the procedure was called. Both procedure statements and the use of the procedure identifier as a function designator are monitored. Standard procedures cannot be monitored.

Example:

```
**LINE 250 CALL TO ALPHA FROM LINE 512
```

The procedure ALPHA (declared in line 250) was called from a statement on line 512.

The overlay declaration provides the capability of overlaying blocks so that a program may be executed in less memory than would otherwise be needed. From the point of view of code generation, the only effect of an overlay declaration is to alter the memory allocation scheme of the translated statements.

Prior to execution, the parts of an overlay program are linked by the loader and placed on a mass storage device or tape file in their absolute form; thus no time is required for linking at execution time. Overlays in a program do increase execution time, however, because each time an overlay is to be executed, it must first be loaded into memory.

The C=2 option must be specified on the ALGOL control statement (see section 6.1.2) when overlays are to be generated.

10.1 OVERLAY DECLARATIONS

10.1.1 SYNTAX

< primary level > ::= < digit > | < digit > < digit >

< secondary level > ::= < digit > | < digit > < digit >

< overlay parameter part > ::= (< external identifier > , < primary level > , < secondary level >)

< overlay declaration > ::= overlay < overlay parameter part >

< declarative comment > ::= comment < overlay declaration > ;

10.2 EXAMPLE

comment overlay (ovfile,1,0);

10.3 SEMANTICS

Primary level can be any positive integer number in the range 1 through 63. Secondary level can be any positive integer number in the range 0 through 63. The main overlay is defined by the system and contains the outermost block of the program as well as the run time system. Therefore, the user must not declare the main overlay.

In any case, a maximum of three overlay levels, including the main overlay, is allowed. The external identifier is the file name to which the loader is to write the absolute program. The same file name must be used in all overlay declarations. Overlay declarations apply only to blocks and procedure bodies (which always constitute a block), thus the comment containing the overlay declaration must appear after the begin of the block to be overlaid.

The overlay declarations in a program must conform to the following rules:

1. A secondary overlay must be a block nested in a primary overlay block.
2. An overlay cannot be a block nested in an overlay block of the same level.

A primary overlay is defined by setting

- a) $\langle \text{primary level} \rangle \neq 0$ and
- b) $\langle \text{secondary level} \rangle = 0$.

A secondary overlay is defined by setting

- a) $\langle \text{primary level} \rangle$ equal to the primary level of the primary overlay it is attached to and
- b) $\langle \text{secondary level} \rangle \neq 0$

For example:

overlay (myfile,1,0) defines a primary overlay and
overlay (myfile,1,1) defines a secondary overlay of the preceding primary

All primary overlays are stored at the same point immediately following the main overlay (0,0). Therefore, the loading of a primary overlay will destroy the preceding primary overlay.

A secondary overlay is stored immediately following its primary overlay. Again, the loading of a secondary overlay will destroy the preceding secondary overlay.

10.4 RESTRICTIONS

A code procedure cannot contain overlay declarations.

10.5 EXAMPLES

Example 1 :

This example demonstrates a simple, direct way of programming overlay structures and produces the most efficient execution time overlay handling.

Di is a declaration

Si is a statement

begin comment block 1;

D1; S1;

begin comment block 2;

comment overlay (F1, 1, 0);

D2; S2;

begin comment block 3;

D3;

comment overlay (F1,1,1);

if expr1 then goto L;

S3

end of overlay (1,1);

begin comment block 4;

comment overlay (F1,1,2);

D4;S4

end of overlay (1,2)

end of overlay (1,0);

L: begin comment block 5;

comment overlay (F1,2,0);

D5; S5;

if expr2 then

```

begin comment block 6;

    comment overlay (F1,2,1):

        D6;S6

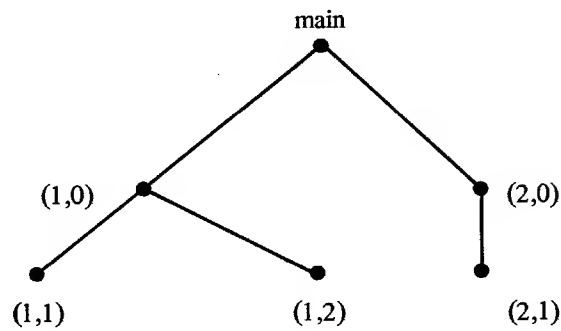
    end of overlay (2,1)

end of overlay (2,0)

end

```

The preceding program structure consists of two primary level overlays; the first contains two secondary level overlays; the second refers to only one secondary level overlay.



The order of execution of the blocks and the structure of memory at execution time are as follows:

Execution Phases

Memory Map for Generated Code

Execution of main is started.

main

Block 2 is initiated and overlay (1,0) is loaded.

main
(1,0)

Block 3 is initiated and overlay (1,1) is loaded.

main
(1,0)
(1,1)

If expr1 is true, then block 3 execution is ended and block 5 is initiated causing overlay (2,0) to be loaded.

main
(2,0)

Otherwise block 3 is executed to its end, and block 4 is started, causing overlay (1,2) to be loaded.

main
(1,0)
(1,2)

At the end of block 4, block 5 is started, causing overlay (2,0) to be loaded.

main
(2,0)

If expr2 is true, then block 6 is initiated which results in the loading of overlay (2,1), otherwise the program is ended.

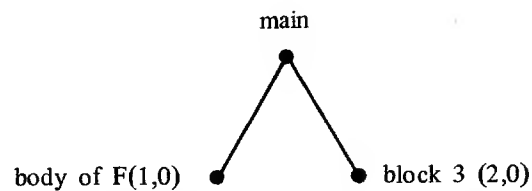
main
(2,0)
(2,1)

Example 2 :

This example demonstrates a more sophisticated and time consuming use of overlays.

```
begin integer J;  
  
    procedure F(N); integer N;  
  
    begin comment procedure body in block 2;  
        comment overlay (file,1,0);  
        J:=N/2;  
        if J > 0 then F(J-3);  
    end of procedure body and overlay (1,0);  
  
    begin comment block 3;  
        comment overlay (file,2,0);  
  
        integer K;  
        K:=5;  
        F(K**2)  
    end of overlay (2,0)  
  
end
```

The program consists of two primary level overlays.



Procedure F is declared in main. The procedure body, however, is a primary level overlay; it will be loaded only when procedure F is executed.

The order of execution of the blocks and the structure of memory at execution time are as follows:

Execution Phases

Memory Map for Generated Code

Execution starts with main, then overlay (2,0) is loaded. A call is made to procedure F.

main
(2,0)

Procedure body is loaded overlaying block 3.

main
(1,0)

When executing $J:=N/2$ in procedure F, block 3 is loaded, overlaying the procedure body to evaluate the called-by-name formal parameter N whose actual parameter originates from block 3.

main
(2,0)

Overlay (1,0) is then reloaded to continue procedure execution. Further recursive executions of F will only refer to elements local to main and the procedure body, and will therefore cause the loading of no other overlays.

main
(2,0)

When $F(K**2)$ is evaluated, return to block 3 is executed and overlay (2,0) is loaded one more time.

main
(2,0)

11.1 VIRTUAL ARRAYS AND S OPTION

The user controls allocation of arrays by a combination of the S option on the ALGOL control statement and the virtual comment delimiter (see chapter 6). ECS/LCM is used more effectively when the S=1 option is selected and arrays are declared as virtual arrays.

A virtual array must be referenced only as a whole, not by individual elements; it can only be used as a parameter to the MOVE procedure (see section 3.5) or other procedures. That procedure can be used to copy a virtual array to a regular central memory array, where it can be modified, or to copy a regular central memory array to a virtual array after modification.

The S=1 option allocates virtual arrays to ECS/LCM. S=0 allocates virtual arrays to central memory (or SCM). The S=2 option allocates all arrays to LCM; this option is only valid under the SCOPE 2 operating system.

When a virtual comment directive is encountered, all arrays in the immediately following array declaration up to the next semicolon are virtual. When a virtual array is used as a formal parameter, the virtual comment directive must precede the array specification in the procedure heading.

A virtual array cannot be an own array.

A virtual array used as a formal parameter cannot be specified to be an array called by value.

In the following example A is declared to be a virtual array and B is declared to be a normal array. The sole use of a virtual array is as a parameter to a procedure, either a normal procedure (shown by P(A)) or the special procedure MOVE. In this example, the normal array B is moved to the virtual array A.

A and B are allocated to CM/SCM or ECS/LCM depending upon the selection of the S option.

```
begin comment virtual; array A [ 1:5, 1:5 ];
      array B [ 1:5, 1:5 ];
      procedure P(A); comment virtual; array A;
          begin
              MOVE (B,1,1,A,1,1,25);
          end;
      P(A)
end
```

11.2 DATA STACKS IN ECS/LCM

Programs which make heavy use of procedures, especially recursive procedures, execute in a reasonably small central memory field length when the G option is selected.

During execution, every accessible set of data related to currently active procedures is retained in central memory. All inaccessible sets of data are swapped out to ECS/LCM. Deep recursions can take place then, as well as long procedure call chains. The only limit is the ECS/LCM field length.

Any ECS/LCM parity errors that occur during a swap are trapped. A message is issued to the dayfile and the job is terminated.

11.2.1 RESTRICTIONS

The G option can be used only when ECS/LCM is available.

Since some checking is performed at run time for swapping purposes, a program which makes heavy use of procedure calls and is executed in optimized mode can be expected to run slightly slower with the G option on, whether or not any swapping occurs. Input/output procedure calls are not slowed by the G option.

No dynamic dumps with traceback are produced when the G option is used, since no pointer relocation takes place for a dump of an ECS/LCM stack. Calls for a dynamic dump with traceback are automatically reduced to a dump of the current block, and the following warning message is issued: SWAPPING OPTION. REDUCED DUMP. Post-mortem traceback dumps are produced even when swapping has occurred; reverse swapping is performed to enable such dumps.

11.2.2 PROCESSING OF DATA STACKS

Swapping takes place when a procedure call is executed. A new set of statically accessible procedures is defined according to the static program structure. Sets of data related to a procedure which is no longer accessible after the call are eliminated from central memory.

Whenever a procedure is invoked, its parameter descriptors, display, and header are put into the next available position in the central memory stack. Swapping conditions are then tested, and if no swapping is to be done, execution continues.

If swapping conditions are satisfied, the CM/SCM stack area for the procedure to be swapped is copied to the next available location of the ECS/LCM stack. The CM/SCM stack is then consolidated by writing over the swapped area. The ECS/LCM address of the swapped stack area is kept in the CM/SCM stack area of the procedure causing the swap.

At the end of procedure execution a reverse swapping operation is performed to restore the CM/SCM stack to its previous contents. The CM/SCM stack for that procedure is overwritten by any ECS/LCM stack it was pointing to, and the next available location for either stack is updated.

11.2.3 SWAPPING CONDITIONS

Two conditions must be satisfied before a swap can occur. One involves the level of the procedure to be swapped, and the other involves the types of the parameters of the call.

A procedure is a candidate for swapping when the procedure it calls is at the same or an inferior static level, and all the parameters of the call are such that there is no possible access to the calling procedure.

Four types of parameters are considered:

1. Simple call by value

The value is assigned to the formal parameter in the calling procedure at call time; therefore, simple formal parameters called by value allow swapping.

2. Simple call by name

The correspondence between a called by name formal parameter and its actual parameter is performed inside the called procedure body every time the formal parameter is referenced. Simple formal parameters called by name allow swapping when their corresponding simple actual parameters are not local to the calling procedure. Actual expressions corresponding to simple called by name parameters prohibit swapping since the expression might contain procedure calls and bring about access to any level. Actual expressions are costly when called by name. Unless the actual expression is intended to produce side effects, the corresponding formal parameter should always be specified by value.

3. Arrays

Value assignment to a called by value formal array is performed inside the procedure body at call time, so formal arrays called by value allow swapping. The formal-actual correspondence for a called by name array is similar to the one for called by name simple variables, so formal arrays called by name allow swapping when corresponding actual parameters are not local to the calling procedure.

4. Strings, Labels, Switches, and Procedures

Because strings may not be used except in an environment exterior to ALGOL (such as the ALGOL ECS/LCM stack) and because labels, switches, and procedures as parameters can access any procedure level, any parameters specified as strings, labels, switches, or procedures will prohibit swapping.

11.2.4 EXAMPLES

11.2.4.1 ACKERMAN'S FUNCTION

```
integer procedure ACKER(M, N); integer M, N;
```

```
ACKER:=if M=0 then N+1
```

```
      else if N=0 then ACKER(M-1, 1)
```

```
      else ACKER(M-1, ACKER(M, N-1));
```

The structure of the procedure ACKER is one of the most recursive structures found in ALGOL programming. It always allows swapping even though the actual parameters bring about recursive calls.

For example, the call ACKER(3,5) would result in 42,438 recursive calls and would use approximately 20 central memory words for its stack if the G option was used. Without the G option, 15,000 words would be required for the central memory stack.

11.2.4.2 MUTUAL RECURSION AND ARRAYS IN LARGE MEMORY

```

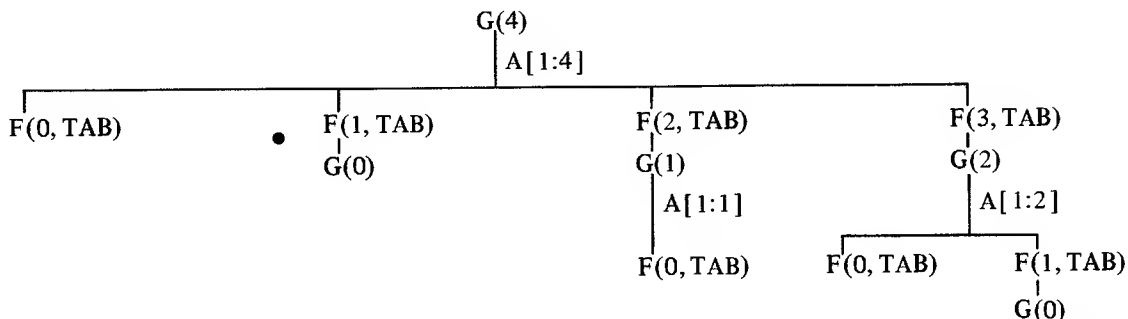
integer array TAB [0:1000];
integer procedure F(N,T);value N;integer N; integer array T;
begin integer J;
  if N=0 then F:=T[0]
    else begin for J:=1 step 1 until N do T[0]:=T[0]+T[J];
           F:=G(N-1);
        end
  end;
integer procedure G(N);value N;integer N;
begin integer I;
  if N=0 then G:=TAB[0]
    else begin array A[1:N];
           for I:=1 step 1 until N do
             TAB[I]:=F(I-1,TAB)+TAB[I];
           G:=TAB[1]
        end
  end
end

```

Procedures F and G call each other recursively and have side effects on the array TAB. The following chain of procedure calls results from the call G(4):

G(4),F(0,TAB),F(1,TAB),G(0),F(2,TAB),G(1),F(0,TAB),F(3,TAB),G(2),F(0,TAB),F(1,TAB),G(0)

This chain has the following tree structure:



The following diagrams represent the memory stack organization when the procedure calls chain is at its longest — when G(4), F(3,TAB), G(2), F(1,TAB), and G(0) are active.

G=0, S=0		G=0, S=2 [†]	
CM/SCM	ECS/LCM	SCM	LCM
TAB	--	TAB	A[1 :4]
G(4)	--	G(4)	A[1 :2]
A[1 :4]	--	F(3, TAB)	--
F(3, TAB)	--	G(2)	--
G(2)	--	F(1, TAB)	--
A[1 :2]	--	G(0)	--
F(1, TAB)	--	--	--
G(0)	--		
--	--		

G, S=0		G, S=2 [†]	
CM/SCM	ECS/LCM	SCM	LCM
TAB	G(4)	TAB	A[1 :4]
G(0)	A[1 :4]	G(0)	G(4)
--	F(3, TAB)	--	F(3, TAB)
--	G(2)	--	A[1 :2]
--	A[1 :2]	--	G(2)
--	F(1, TAB)	--	F(1, TAB)
--	--	--	--

[†]Applies only when under SCOPE 2.

The compiler optionally performs four kinds of optimization while the run time system is itself organized to perform optimally at all times. In addition, the compiler is provided with a set of standard vector functions which allow the user to optimize source programs at the algorithm level and benefit from certain hardware features.

12.1 MACHINE INDEPENDENT OPTIMIZATION

Machine independent optimizing involves subscript variable handling, the calling of procedures and the handling of formal variables, and special treatment of the procedures INPUT and OUTPUT.

12.1.1 SUBSCRIPTS

The source program is delimited into nodes, consisting of statements and blocks, in which code is to be generated so that as many subscript computations as possible share common code while still retaining the explicit statement ordering. The largest gains come from the for statement in which as many hidden computations as possible are removed from the loop. It is necessary, however, to re-compute the values resulting from the removed sequences when any dependent operand is changed. Such changes can occur at assignment statements and in procedure calls due to assignment to actual parameters or by global side-effect. For dealing with procedure calls, a side-effect table is constructed before code generation starts which is consulted after every procedure call in a for-statement.

12.1.2 PROCEDURES

Procedure calling can be time-consuming at run time in ALGOL because of the necessity to check the formal specifications against the actual parameters. The optimizing of this process involves an examination of all the calls to any procedure to determine parameter correctness. Simultaneously the set of all actual parameters corresponding to each formal parameter is found, permitting special case code to be generated in the procedure body for all call-by-name formal parameters. The ALGOL Report states that the procedure bodies are to be considered as being inserted in the program at calls with the actual parameters replacing the formals, as in macros, and of course this would lead to optimal code at each call. This compiler does not do that; it provides one copy of the code of each declared procedure but generated so that the actual parameters of the calls are all treated in the least general manner which accommodates them all. This is not true of pre-compiled procedures in which each call-by-name formal parameter is given the most general treatment.

12.1.3 SIMPLE FORMATTED INPUT/OUTPUT

The formatted input/output procedures INPUT and OUTPUT are the most commonly used input/output routines and the compiler examines their calls to determine if the parameters obey certain restrictions. If they do then calls are generated to special library routines which execute faster than the general case. The restrictions which will produce these calls are that the format string include no items with insertion sequences (chapter 3, section A.1.3.2) and that the data parameters be simple local variables or arrays but not subscripted variables or formal parameters.

12.2 VECTOR FUNCTIONS

These procedures are available in the standard library and are designed to take advantage of the hardware instruction stack. For machines without a stack, the gain from using the vector functions over programmed loops performing the same operations is about a factor of 5. For machines with a stack, the gain is of the order 10 or more. In all cases, the vectors concerned need to be of size 10 or greater before the gains are appreciable, owing to initialization performed within the functions. The vector functions are described in appendix D.

12.3 EFFICIENT PROGRAMMING FOR RUN-TIME PERFORMANCE

ARRAYS: Within each block, arrays of the same size and type should be declared in a single declaration.

PROCEDURES: Although the ALGOL Report requires agreement in type between formal and actual parameters, ALGOL 4 does not. When real actual parameters correspond to integer formal parameters, conversion takes place, resulting in less efficient run-time performance than when types agree. When integer actual parameters correspond to real formal parameters, however, there is no difference, since no conversion takes place.

PROCEDURES INPUT AND OUTPUT: The format parameter should be a literal string in which numeric items do not contain insertion sequences. The data parameters should be local simple variables and arrays.

GENERATED MACHINE CODE: It is not possible to influence this directly but the code is obviously a function of the preceding optimizations.

THE VECTOR FUNCTIONS: When the program is handling vectors and arrays these functions should be employed wherever possible in place of programmed loops. Overall gain in total program execution time of a factor of 10 is possible with appropriately structured algorithms in suitable programs on machines with an instruction stack.

LCM/ECS USAGE: Arrays situated in SCM on the CDC CYBER 70 Model 76 are faster to access than those in LCM. For optimal usage of LCM use should be made of the virtual array concept for arrays situated in LCM while declaring only as many normal arrays in SCM as needed for working purposes. This requires careful programming to keep track of the contents of work arrays at any time. On other machines only virtual arrays can be in ECS and only considerations of size dictate its use (see chapter 11).

EXECUTION TIME ABNORMAL TERMINATION DUMP 13

Upon detection of a fatal error during program execution, the ALGOL execution time routines perform the following actions:

Empty all output format areas onto their associated files.

Print the appropriate diagnostic on the standard output file (channel 61).

Print a structured dump. The amount of information given here is determined by the execution time option parameter D (see chapter 8).

See chapter 15 for a list and description of the object time diagnostics.

13.1 STRUCTURED DUMP

The structure dump traces back the execution path from the point where the error occurred through the block structure to the entry point of the program. The information relevant to the program is selected according to the D option.

The following information is available:

The line number at which the error occurred.

The line number and name in which each active block was declared. This name could be the procedure name, the code procedure identifier, the external procedure identifier or the standard procedure name if the block is a standard procedure.

Code procedures are always dumped in octal.

13.2 ENVIRONMENTAL INFORMATION

Environmental information consists of values of formal (by value) or local variables belonging to the block currently being dumped. Formal parameters appear only if the particular block is a procedure.

Simple local variables and simple formal parameters are represented by their values.

Arrays may also be dumped according to the D option.

For Boolean variables the values *T* (true), and *F* (false) will be printed.

When no values have been assigned to a variable, the information dumped is irrelevant.

13.3 CROSS-REFERENCE LISTING

The cross-reference listing, requested with the R option of the ALGOL control statement, is a useful debugging aid in conjunction with error messages.

The format of the cross-reference listing follows the program structure, beginning with a list of all standard procedures used in the program and the line numbers where each is referenced. A list of all blocks and procedures used in the program follows in order by block number.

The listing for each block consists of a header line, a list of declarations, a list of references, and an end line.

The header line contains the block number, the level of nesting, and the line number of the first line of the block.

All identifiers declared in the block appear in alphabetical order. For each identifier, the following information is printed:

Name of identifier

Type: real, integer, Boolean

Kind: array, procedure, switch, label

Line of declaration

Number of dimensions for an array

Number of formal parameters for a procedure

Line numbers indicate where each declared identifier is referenced. Following a line number, the letter P indicates the variable is used as an actual parameter; the letter S indicates a value is assigned to that variable at that line.

The end line contains the line number of the last line of the block.

The listing for a procedure consists of a header line, a list of all identifiers specified in the formal parameter list, a list of references for the formal parameters, a list of declarations, a list of references for the procedure body, and an end line.

The header line is the same as the header line for a block.

The list of all identifiers specified as formal parameters appear in alphabetical order. For each, the following information is printed:

Name of identifier

Type: real, integer, Boolean

Kind: array, procedure, switch, label

Line of specification

The list of references indicates the line numbers where each formal parameter is referenced. The letters P and S have the same meaning as for a block.

The listing for the procedure body itself is the same as the listing for a block. It is followed by an end line containing the line number of the last line of the procedure.

14.1 OBJECT PROGRAM AND STACKS

14.1.1 RUN-TIME SUPERVISORY PROGRAM

A Run-Time Supervisory program, ALGORUN, controls the object-time execution of an ALGOL program. In this function it acts as the interface between the compiled program and the operating system. Certain functions are shared between the compiled program and ALGORUN (e.g., stack handling), whereas others (e.g., overlay handling) are solely the province of the latter.

Calls to the Run-Time System and Library are treated as references to externals during compile time and are satisfied by the loader at load time.

14.1.2 OBJECT CODE STRUCTURE

The object-time form of each main program or code procedure consists of a program block and several labeled common blocks. The program block contains the block map and the code for the program itself. The labeled common blocks contain constants, strings, label descriptors, procedure descriptors, the own variables, and a special parameter area. The object code is generated forward, from the first begin to the last end, and forms the binary relocatable file.

In overlay mode, the program block is organized according to the overlay structure specified, each overlay being preceded by the overlay directive to the loader. The labeled common blocks are attached to the main overlay.

14.1.3 LIBRARY SUBPROGRAMS

The Library subprograms obey the same structural rules as a binary subprogram generated by the compiler. Each standard procedure is represented in the library by one subprogram. In overlay mode, a required subprogram is attached to the earliest overlay that requests it.

14.1.4 SCALAR SPACE

The "scalar space" of a block includes storage for all those items which can be quantified at compile time, such as simple variables declared in the block, array descriptors, dope vectors and various compiler generated locations. It does not include storage for arrays.

14.1.5 PROGRAM DEPTH AND PROGRAM LEVEL

The “program depth” of a block is the number of nested procedure declarations within which the block is contained.

A “program level” is the largest set of blocks of the same depth, such that one block of the set encloses all of the other blocks (if any) of the set. The enclosing block is called the “first block” of the set and has the property that its depth is one greater than the depth of its own enclosing block (if any). The program level is the unit of allocation of scalar space for blocks, such that the relative positions of scalar space for all blocks within a program level are determined at compile time. The address to which the variables in a program level are relative is called the “relocation base”.

“Program level zero” is the unique program level whose constituent blocks have a zero program depth. The “first block” of program level zero is the outermost block of the main program.

14.1.6 OBJECT TIME STACKS

According to the rules of the ALGOL language, a variable is active (available for reference) in any block to which it is local or global. A variable is local to the block in which it is declared and global to the sub-blocks within the block in which it is declared.

Depending on the block structure and the variables declared at each level, not all variables are active at the same time. The object programs produced by ALGOL overlap variables which are not simultaneously active. The overlap process is described below.

During execution of an object program, all variables are contained in variable-length memory stacks consisting of 60-bit entries, one or more pertaining to each variable. Since the stacks include only active entries, their sizes fluctuate.

The SCM/CM stack is located in SCM or CM and contains the scalar space for program levels, the actual parameters and return information for procedures and the SCM/CM-resident arrays.

The LCM/ECS stack (if it exists) is located in LCM or ECS. It only contains the virtual arrays (see Chapter 11). Non virtual arrays may be also allocated in LCM under the S storage option (see Chapter 6).

14.1.7 REFERENCING THE STACKS

Every reference to simple variables in the SCM/CM stack is generated as the relative position of the variable in its program level plus an index register which contains the appropriate relocation base.

When a call to a procedure is made, the environment of the procedure has to be established because this call constitutes an entry into a new program level. The entry into the new program level is achieved by assigning the first available (inactive) position in the stack to the relocation base for the new program level and by requesting the necessary amount of stack for the scalar space of the complete program level (i.e., including all its blocks). Also a “display” is built into the stack, containing the relocation bases for all the statically enclosing program levels.

When a new block is entered, its position within the program level in which it is contained is known and its scalar space requirements have already been reserved. It only remains to physically record its stack limit before handing control to the block’s code.

Array storage in SCM/CM is physically assigned starting at the first available position in the SCM/CM stack after the allocation for a complete program level, and its stack limit is recorded in the block where the arrays are declared. Arrays are indirectly addressed through array descriptors and dope vectors contained in the scalar space of their declaring block.

Arrays in LCM/ECS are also indirectly addressed through array descriptors and dope vectors in the scalar storage of their declaring block (located in SCM/CM stack). Array storage in LCM/ECS is assigned starting at the first available position in the LCM/ECS stack and its stack limit is also recorded in the declaring block in SCM/CM stack.

When a block is exited, the space in the stacks (SCM/CM and LCM/ECS) occupied by its local arrays is released, so that it may be allocated by a new stack request, however, the amount of SCM/CM stack occupied by its scalar space can only be utilized by a parallel block in the same program level.

When a procedure is exited, the scalar space of the complete program level is then released.

A go to reference from one block in a nest to an outer one results in an exit from that block and from all of the blocks up to but not including the referenced block. Thus the effect is to change the environment of the active variables to be only those local or global to the referenced block.

14.1.8 OWN VARIABLES

All own quantities are assigned entries in the labeled common block area. Own variables are treated as global in definition (local to the whole program), though they are local only to the block in which they are declared, just like other variables.

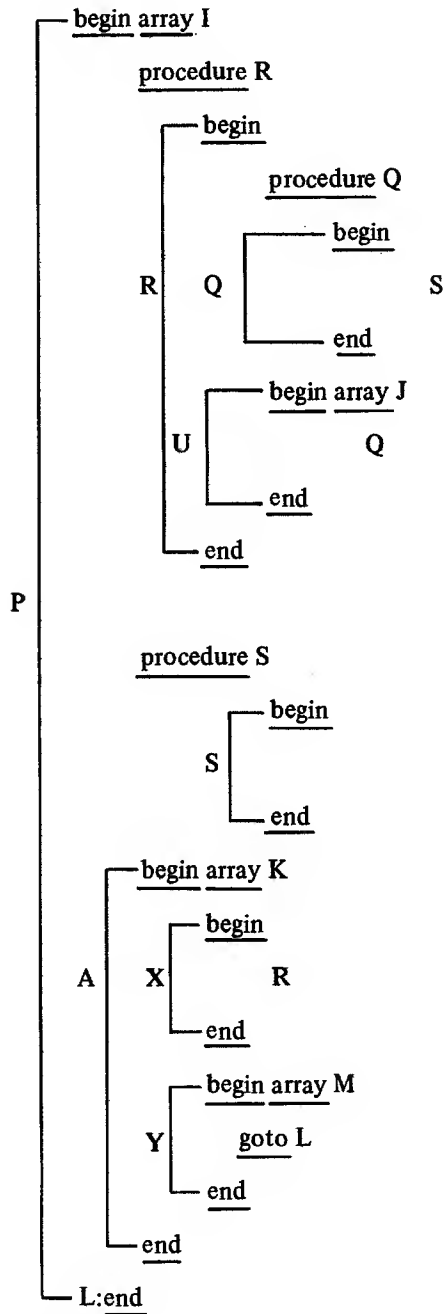
14.1.9 STACK LISTING

The compiler assigns a block number to each block in the program and constructs the block map which contains information about the block and procedure structure of the program for use during traceback and procedure entry.

The object program controlling system includes a routine which produces the active contents of the stack in a meaningful format upon abnormal object program termination. This structured dump may also be called by the procedure DUMP.

14.1.10 EXAMPLE

Consider the following program outline:



Block P is the program itself; block A and procedures R and S are at the same level within P; block U and procedure Q are contained in procedure R at the same level. Blocks X and Y are at the same level within block A.

Block X contains a call to procedure R, block U a call to procedure Q, and procedure Q a call to procedure S; block Y contains a jump to label L within the outermost block. Blocks P, U, A and Y declare respectively arrays I, J, K, and M. The changes in the stack and in the displays are shown in figure 14-1.

Following the entry into the program, the stack is already reserved for blocks A, X and Y, although it is not yet used. The array I follows in the stack. The display at P indicates that only program level P is accessible, through its relocation base address p.

On entry to block A, array K is allocated after I in the stack. When procedure R is called, space for all of its program level is reserved after K and the display in R indicates that both p and r are accessible. The compiler knows which blocks are accessible (as opposed to program levels, which are dynamic) and since space for them is always prereserved, it is unnecessary to include block information in the displays.

On entry to U, array J is placed after U in the stack. When procedure Q is called, its display contains p, r and q. However, when procedure S is entered, its display contains only p, and although R, U, J and Q are still in the stack, they become inaccessible.

On exiting S and releasing its stack space, R, U, J and Q become active automatically, since the display at Q already exists. When eventually R is exited, the display at P will indicate that only p is accessible.

On entering block Y, the compiler knows that space is already available and only allocates stack for array M. This exactly follows the rules described in the ALGOL 60 Revised Report concerning the accessibility of variables during and after return from a procedure call.

14.2 STACK ENTRIES

14.2.1 VALUE OF VARIABLES

Simple local variables and simple formal parameters called by value are represented in the stack as follows:

REAL

60-bit entry in standard floating-point format (section 5.1.3, chapter 2)

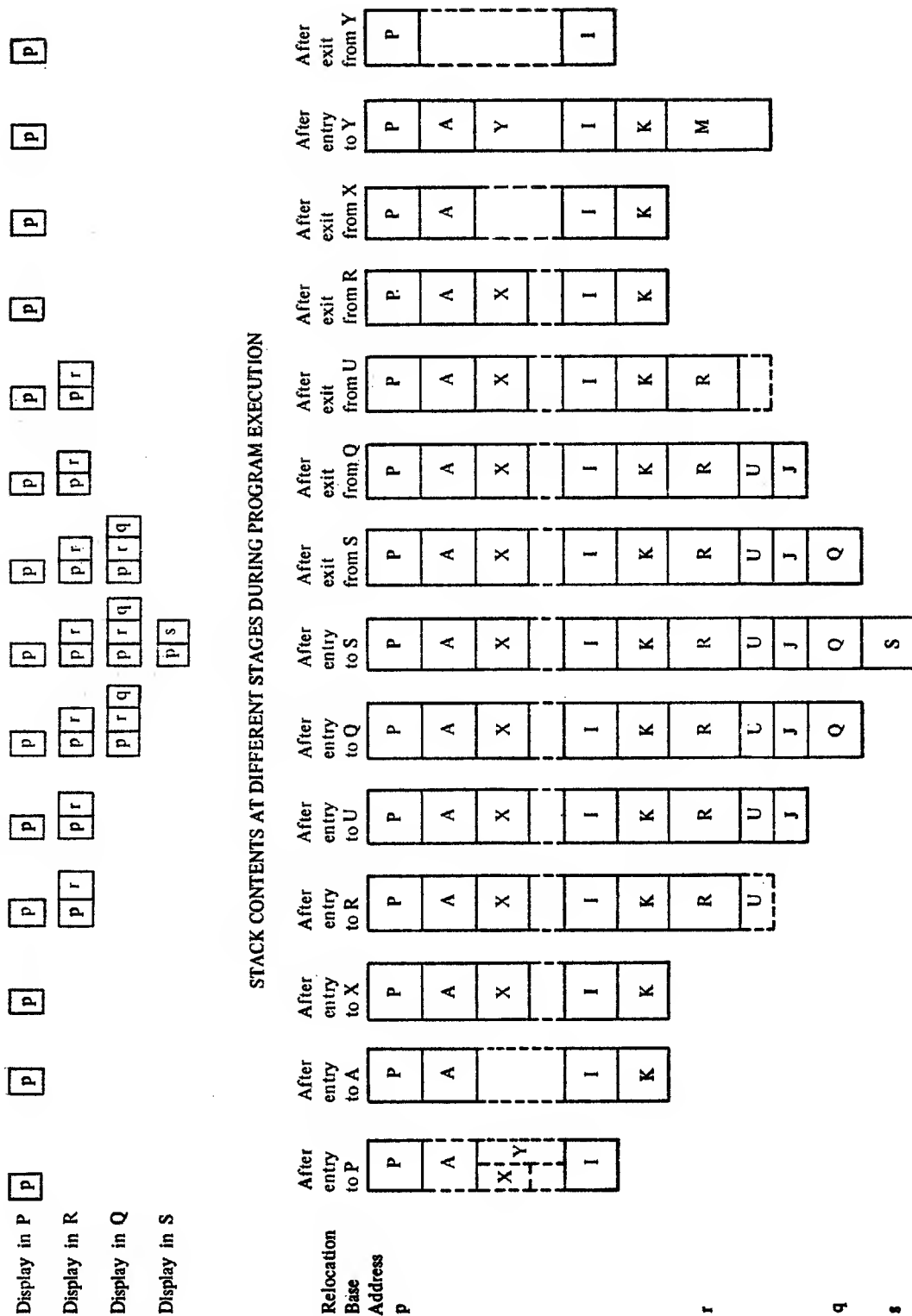
INTEGER

60-bit entry in standard floating point format (section 5.1.3, chapter 2)

BOOLEAN

60-bit entry in which bits 58-0 are irrelevant and bit 59 is set to 1 for true and 0 for false.

Figure 14-1. Sample Stack and Display Contents



14.2.2 DESCRIPTORS OF VARIABLES

All descriptors of variables in the stack have the following general form:

xxxx	xxxxxxx	xxxx	xxxxxxx
$x=0: \langle x \rangle_1 \langle i \rangle_1 \langle 0 \rangle_3 \langle k \rangle_4 \langle t \rangle_3$ $x=1: \langle x \rangle_1 \langle \bar{i} \rangle_1 \langle 7 \rangle_3 \langle \bar{k} \rangle_4 \langle \bar{t} \rangle_3$	address 1	$\langle s \rangle_1 \langle 0 \rangle_4$ $\langle \text{overl} \rangle_7$	address 2

x is the transform bit; if x = 1 no transformation is required and the following 11 bits are complemented

if x = 0 a transformation from real to integer is necessary

i is the expression bit; if i = 1 no expression evaluation is required

if i = 0 expression evaluation is necessary

s is the storage bit for arrays, virtual arrays and subscripted variables

k is the kind as follows:

k	Kind	Possible use
00	Switch	Formal and local
01	String	
02	Label or designational expression	
03	No-type procedure	
04	Type procedure	
05	Array	
12	Virtual array	Formal only
06	Constant	
07	Expression	
10	Simple variable	
11	Subscripted variable	
13	No-parameters, no-type procedure	
14	No-parameters, type procedure	

t is the type of the variable

<u>t</u>	<u>Type</u>	<u>Possible use</u>
0	No type	Formal and local
1	Integer	
2	Real	
3	Boolean	

overl is an index to the overlay table (when necessary)

The interpretation of address 1 and address 2 depends on the kind (k) of the descriptors as explained below.

A stack entry representing an arithmetic value can have a structure which makes it appear to be a descriptor.

14.3 DETAILS OF DESCRIPTORS

The following detailed explanations of the descriptors are ordered according to the kind k. Return information for a procedure call does not have a kind; it is described first.

14.3.1 TERMINOLOGY

All references to the stack in the object program are relative to the beginning of the stack area for a particular program level. When a program level is entered at execution time, the base address of the corresponding stack area is assigned. This absolute base address is the relocation base RRRRRR of the program level (and of all the blocks it encloses). The term program address PPPPPP means an address pointing to a position in the object program. The term stack address AAAAAA means an absolute address pointing to a particular stack entry. The term common address CCCCCC means an absolute address pointing to a particular labeled common area.

14.3.2 DESCRIPTION

Return Information for Expressions	XXXXX	XXX	XXXXXX	XXXXXX
	LINE	OVERL	RRRRRR	PPPPPP

PPPPPP Program address of next instruction following the evaluation of the expression.

RRRRRR Relocation base of the program level where the expression evaluation is requested.

OVERL Index of the overlay where the expression evaluation is requested.

LINE The source line number where expression evaluation is requested.

**Return
Information for
Procedures**

xxxx	xxxxxx	xxxx	xxxxxx
Number of formals	RRRRRR	OVERL	PPPPPP

RRRRRR Relocation base of the program level where the call is made

PPPPPP Program address of next instruction following the procedure call

OVERL Index of the overlay where the call is made

00 Switch

XXXX	XXXXXX	XXXX	XXXXXX
5777	NNNNNN	0000	AAAAAA

NNNNNN Number of elements in the switch list

AAAAAA Stack address for the first element in the list

The descriptions of the labels or designational expressions (see kind 02 below) which constitute the switch list is as follows:

< Designational expression of the n^{th} switch element >₆₀

< Designational expression of the $(n-1)^{\text{th}}$ switch element >₆₀

aaaaaa < Designational expression of the 1st switch element >₆₀

01 String

XXXX	XXXXXX	XXXX	XXXXXX
5767	< C > ₁ < T > ₁ < X > ₆ < N > ₁₀	0000	CCCCCC

< C > format flag ; if < c > = 1, the string has been analyzed and can be used as a format string

if < c > = 0, the string must be analyzed

< T > translation flag ; if < T > = 1, the string has been translated

< X > X Replicator count (63 maximum)

< N > Number of characters in the string (maximum of 1023 characters)

CCCCCC Address of first word of the string in a labelled common

02 Label

XXXX	XXXXXX	XXXX	XXXXXX
5757	AAAAAA	OVERL	PPPPPP

AAAAAA Block header's address (in the stack) of the block where the label is declared

OVERL Index of the overlay where the label is contained

PPPPPP Program address of the instruction corresponding to the label

**02 Designational
Expression**

xxxx	xxxxxxx	xxxx	xxxxxxx
5757	AAAAAA	OVERL	PPPPPP

AAAAAA

Block header's address (in the stack) of the block where the designational expression is contained

OVERL

Index of the overlay which contains the expression

PPPPPP

Program address of the code which evaluates the expression and jumps to the resulting address

**03 No-type
procedure**

xxxx	xxxxxxx	xxxx	xxxxxxx
5747	RRRRRR	OVERL	PPPPPP

and

**04 Type
procedure**

xxxx	xxxxxxx	xxxx	xxxxxxx
573t 204t	RRRRRR	OVERL	PPPPPP

RRRRRR

Relocation base of the program level where the procedure is declared

OVERL

Index of the overlay where the procedure is declared

PPPPPP

Program address of the code for the procedure

05 Array

xxxx	xxxxxxx	xxxx		xxxxxxx
572t 205t	AAAAAA	S	0000	DDDDDD

AAAAAA

Stack address where the quantity FWA-LBE is contained

FWA is the base address of the array elements in stack

LBE is the lower bound effect for the array

DDDDDD

Stack address of the dope vector (see below)

S

Flag indicating if the array is in LCM (=1) or in SCM (=0) (Bit 29)

The elements of an array are assigned after the last location reserved for the program level where the array's declaring block is contained. Own arrays are handled in the same way, except that their elements are assigned among the own variables. The elements of an array called by value are copied (and transformed as necessary) to a position after the last location reserved for the program level from where the array is called. Array elements are stored consecutively in ascending memory locations. The order of storage is with the leftmost subscript varying most rapidly, and the rightmost subscript varying least rapidly.

Example:

array A[1:3,1:2]

The elements of this array are stored as follows:

A[1,1]	A[1,2]	A[2,1]	A[2,2]	A[3,1]	A[3,2]
↑				↑	
FWA+0				FWA+5	

The dope vector for the array declaration

array A [L₁:U₁, L₂:U₂, ..., L_n:U_n] is:

	<		C _n	>	60	}	Dope Vector		
	<		C _{n-1} * C _n	>	60				
								
	<	C ₂ * C ₃ *	* C _{n-1} * C _n	>	60				
	<	LENGTH = C ₁ * C ₂ *	* C _{n-1} * C _n	>	60				
DDDDDD→	<	Lower Bound Effect = LBE		>	60				
	<	Number of dimensions = n		>	60				
This part of dope vector exists only when array bound check- ing is required	<		L ₁	>	60			}	Size = 3n + 2 words
	<		U ₁	>	60				
	<		L ₂	>	60				
	<		U ₂	>	60				
	<	L _n	>	60				
	<		U _n	>	60				
								
								

AAAAAA→ < FWA - LBE > 60

where $C_i = U_i - L_i + 1$

and

$$LBE = (((\dots (L_1 * C_2 + L_2) * C_3 + L_3) * \dots) * C_n + L_n$$

The address of any element A [i_1, i_2, \dots, i_n] is calculated as follows:

$$\text{address} = FWA + (((\dots i_1 * C_2 + i_2) * C_3 + i_3) * \dots) * C_n + i_n - LBE$$

For example, the declaration array A [1:3, 2:5] has a dope vector:

```

          < 4 > 60
          < 12 > 60
ddddd→ < 6 > 60
          < 2 > 60
          < 1 > 60
          < 3 > 60
          < 2 > 60
          < 5 > 60

```

and a descriptor

```
< 5725  aaaaaa  0000  ddddd > 60
```

where aaaaa→ < FWA - 6 > 60

06 Constant

xxxx	xxxxxx	xxxx	xxxxxx
$\left\{ \begin{array}{l} 571\bar{t} \\ 206t \end{array} \right\}$	000000	0000	CCCCCC

CCCCCC

Common address at which the constant is found.

07 Expression

xxxx	xxxxxxx	xxxx	xxxxxxx
$\left\{ \begin{array}{l} 770t \\ 007t \end{array} \right\}$	RRRRRR	OVERL	PPPPPP

RRRRRR

Relocation base of the program level where the expression is situated.

OVERL

Index of the overlay containing the expression

PPPPPP

Program address of the code that evaluates the expression

10 Simple Variable

xxxx	xxxxxxx	xxxx	xxxxxxx
$\left\{ \begin{array}{l} 567t \\ 210t \end{array} \right\}$	000000	0000	AAAAAA

AAAAAA

Stack address of the variable

11 Subscripted Variable

xxxx	xxxxxxx	xxxx	xxxxxxx
$\left\{ \begin{array}{l} 766t \\ 011t \end{array} \right\}$	RRRRRR	S OVERL	PPPPPP

RRRRRR

Relocation base of the program level where the code to evaluate the address is found

OVERL

Index of the overlay containing the code

PPPPPP

Program address of the code which computes the address

S

Flag indicating if the variable is in LCM (=1) or SCM (=0) (Bit 29)

12 Virtual Array

xxxx	xxxxxxx	xxxx	xxxxxxx
$\left\{ \begin{array}{l} 565t \\ 212t \end{array} \right\}$	AAAAAA	S 0000	DDDDDD

See chapter 11 for differences between this kind and kind 05.

AAAAAA,
DDDDDD, and S

are as described for kind 05

**13 No-parameters
No-type procedure**

xxxx	xxxxxxx	xxxx	xxxxxxx
$\left\{ \begin{array}{l} 7647 \\ 0130 \end{array} \right\}$	RRRRRR	OVERL	PPPPPP

RRRRRR

Relocation base of the program level where the expression's code to call the procedure is found.

OVERL

Index of the overlay containing the expression's code

PPPPPP

Program address of the code where the procedure call is made.

**14 No-parameters
Type procedure**

xxxx	xxxxxxx	xxxx	xxxxxxx
$\left\{ \begin{array}{l} 763\bar{t} \\ 014t \end{array} \right\}$	RRRRRR	OVERL	PPPPPP

RRRRRR, OVERL and PPPPPP are as described for kind 13.

Two types of diagnostics are issued by the ALGOL compiler system: compile time diagnostics and object time diagnostics. Diagnostics are also issued by the ALGOL Interactive Syntax Checker (ALGEDIT).

15.1 COMPILER DIAGNOSTICS

Every error detected during compilation causes a diagnostic to be printed following the source listing. If the source listing is suppressed, the diagnostics are output to the standard output device. Each source line is assigned a line number, which is printed as part of the source listing, and each diagnostic includes the line number of the source line in error and a summary of the error condition.

The diagnostics are grouped under the following headings:

LEXICOGRAPHIC AND SYNTACTIC DIAGNOSTICS

PRE-SEMANTIC DIAGNOSTICS

SEMANTIC DIAGNOSTICS

PROCEDURE CHECKING WARNINGS (OPTIMIZING MODE)

POST-SEMANTIC DIAGNOSTICS

SYMBOLIC FILE CONSTITUTION WARNINGS — FILE SUPPRESSED

A heading appears only when a corresponding diagnostic occurs.

Compiler diagnostics are either alarms or advisory messages; alarms cause the generation of object code to be suppressed but advisory messages do not.

15.1.1 COMPILER ALARMS

A compiler alarm indicates that a serious error has been found in the source text and causes the suppression of code generation regardless of any user request, although normal compilation and error checking continue until the end of the source text. Source errors cause otherwise legal text to become invalid, but the compiler attempts to localize the effect of each individual error.

The error messages are self-explanatory. Upon compiler detection of an error, the diagnostic is dynamically constructed with additional relevant information (such as identifier name, identifier type, operand name, syntax structure, and line number) inserted in the message. In the following examples the underlined parts indicate the additional

information inserted into the diagnostic:

LINE 123 : ILLEGAL (AFTER , IN ARRAY DECLARATION

LINE 456 : BB SPECIFIED PROCEDURE IS ILLEGAL BY VALUE

LINE 789 : UNRECOGNIZED COMPOUND DELIMITER ~~#SKIP#~~ . . . IGNORED EXCEPT LAST ~~#~~

For a lexicographic error, the illegal character is replaced by a blank acting as a lexicographic separator and the resulting operands are processed according to the current state of the compiler. This often results in a second diagnostic being issued.

Example:

The statement ~~#INTEGER#~~ H\$I;

produces the following diagnostics:

LINE 22 : ILLEGAL CHARACTER \$ OCCURS IN COLUMN 34 . . . ACTS AS A LEXICOGRAPHIC SEPARATOR

LINE 22 : MISSING DELIMITER BETWEEN H AND I . . . FIRST OPERAND DELETED

There are two error messages related to the creation of the dumpfile:

ERROR IN BLOCK TABLE and

NO SYMBOL FILE FOR CODE PROCEDURE

These diagnostics indicate an internal failure in the compiler and should be brought to the attention of Control Data.

15.1.2 RECOVERY MESSAGES

In some cases the error message is preceded by RECOVERY : to indicate a fatal error has been detected and the compiler has attempted to correct the error and continue the compilation. However, it is still considered a fatal error. Furthermore, it is possible that the compiler has guessed incorrectly, in which case the recovery might have provoked additional diagnostics which do not necessarily indicate the presence of other errors. The following are examples of recovery diagnostics:

LINE 15 : RECOVERY : ILLEGAL = AFTER ~~#FOR#~~ REPLACED BY :=

LINE 120 : RECOVERY : ~~#QRRAY#~~ REPLACED BY ~~#ARRAY#~~

15.1.3 ADVISORY MESSAGES

Advisory messages do not necessarily indicate the presence of an error in the source text but provide information which may be useful in detecting errors not recognized as language infringements.

For example:

LINE 55 : ADVISORY : NON-FORMAT STRING

LINE 92 : ADVISORY : DELIMITER(S) BEFORE PROGRAM START
May indicate a missing #BEGIN#

LINE 115 : ADVISORY : ERROR IN DUMP FILE WRITING
Indicates a hardware error; DUMPFIL is suppressed and compilation continues.

The advisory messages, particularly NON-FORMAT STRING, can be a nuisance. They can be suppressed, however, using the option N=0 on the control card. Fatal errors will always be output.

15.1.4 ADVISORY MESSAGES IN OPTIMIZING MODE

When optimization is requested, additional checking is performed on the use of procedures and parameters. In the case of a type or kind mismatch between an actual parameter and a formal parameter, an advisory message is issued.

LINE 104 : ADVISORY : KIND ERROR IN ACTUAL PARAMETER

LINE 189 : ADVISORY : ACTUAL PARAMETER CANNOT CORRESPOND TO LEFT-HAND SIDE
USAGE OF FORMAL.

LINE 237 : ADVISORY : ACTUAL PARAMETER SHOULD BE A PROCEDURE WITH PARAMETERS

In nonoptimizing mode, as well as for the cases listed above, the correspondence between actual and formal parameters is checked at run-time. Any parameter mismatch will only lead to an error if the procedure call is executed.

15.1.5. COMPILER STOP

For a violation of an implementation restriction, a compiler stop occurs. A diagnostic indicating the nature of the error is printed on the output file, followed by a message indicating the compiler stop.

15.1.6 COMPILE ABORT

If the compiler should fail, the following messages will be printed on the dayfile:

COMPILER ABORT IN LINE _____ SCAN _____ .
FROM ALGOL COMPILER _____ .

Indicates an internal error in the ALGOL compiler and should be brought to the attention of Control Data.

15.2 OBJECT TIME DIAGNOSTICS

Upon normal exit from an object program, the contents of all non-empty format areas are output.

Upon abnormal termination, a diagnostic is printed on the standard output channel to indicate the nature of the error, and the contents of all non-empty format areas are output. If asked for at execution time, information which traces the execution path through the currently active block structure and stack information for each block are then printed on the standard output device (see D option, chapter 8).

Object time diagnostics are listed in table 15-1 in alphabetical order.

ERROR and CHANERROR are execution time error trapping procedures which permit the user to regain control after the detection of an error of the same type as the given key (chapter 3, section 3.3). In the table of diagnostics, a key is given for each message where applicable. A call to ERROR or CHANERROR will trap all errors with the corresponding key.

15.3 ALGEDIT DIAGNOSTICS

Error messages displayed by ALGEDIT are listed in table 15-2, in alphabetical order. The commands that can cause the message to appear are listed in the Issued By column.

TABLE 15-1. OBJECT TIME DIAGNOSTICS

Message	Error Key	Chan-Error Key	Significance	Action	Issued By
ALPHA FORMAT ERROR					
ALGOL I/O ERROR nnn ON filename		4	Output value is too large.	Self-evident	Run-time System
		5	I/O error detected on ALGOL channel where nnn is the CRM error code returned by the CYBER Record Manager.	Refer to CRM reference manual.	Run-time System
ARITHMETIC OVERFLOW	I		Evaluation of an expression results in an arithmetic error (e.g., if the operand is infinite). This is the same as a mode 2 error.	Self-evident	Run-time System
ARITHMETIC MODE ERROR	1		More than one mode error has occurred during program calculations (e.g. a MODE control statement may have suppressed earlier detection).	Self-evident	Run-time System
ARRAY TYPE NOT COMPATIBLE	2		Invalid correspondence between actual and formal parameter arrays.	Self-evident	Run-time System
ARRAY DIMENSION ERROR - DIMENSION NO.:	3		Number of dimensions in actual parameter array in procedure call differs from number in formal param- eter array.	Self-evident	Run-time System
ARRAY LOWER BOUND ERROR - DIMENSION NO.: VALUE:	2		Computed element address in an array is not within lower array bound.	Self-evident	Run-time System

TABLE 15-1. OBJECT TIME DIAGNOSTICS (CONT'D)

Message	Error Key	Chan-Error Key	Significance	Action	Issued By
ARRAY UPPER BOUND ERROR - DIMENSION NO.: VALUE:	2		Computed element address in an array is not within upper array bound.	Self-evident	Run-time System
BAD DIAGNOSTIC LIB: ERNUM:			Indicates an internal error in ALGOL.	Inform Control Data	Run-time System
***ERROR IN SYMBOL FILE-			Wrong symbol file used. May be hardware failure.	Notify systems per- sonnel if hardware failure.	Symbolic dump (appears in dump printout)
EXPONENTIAL PARAMETER ERROR	1		Argument of EXP procedure is too large.	Self-evident	Run-time System
FETCH ITEM/LIST TYPE ERROR		3	Attempt to mix variable types on word addressable channel in FETCH ITEM/ LIST call.	Self-evident	Run-time System
FINAL ADDRESS OF MOVE OUT OF ARRAY FIELD - PARAMETER NO:	2		Computed final address is not within array field.	Self-evident	Run-time System
FORMAT ITEM ERROR		4	More characters in expanded format item than permitted in INPUT, OUTPUT, INLIST, or OUTLIST.	Self-evident	Run-Time System
FORMAT MISMATCH	4		Format item and correspond- ing I/O item have incom- patible types or kinds.	Self-evident	Run-time System
FORMAT REPLICATOR ERROR		4	Replicator in call to FORMAT procedure not in proper range.	Self-evident	Run-time System

TABLE I5-1. OBJECT TIME DIAGNOSTICS (CONT'D)

Message	Error Key	Chan - Error Key	Significance	Action	Issued By
FORMAT STRING ERROR		4	Incorrect format string.	Fix faulty I/O statement.	Run-time System
GET ITEM/LIST TYPE ERROR		3	Type of data does not match variable on GETITEM or GETLIST call.	Self-evident	Run-time System
H/V LIM ERROR		6	H LIM AND V LIM arguments L, R, and L', R' out of range.	Self-evident	Run-time System
ILLEGAL CHANNEL NUMBER VALUE:		5	A negative, undefined or infinite channel number was found in procedure CHANERROR.	Self-evident	Run-time System
ILLEGAL KEY VALUE:		5	The key used in procedure ERROR or CHANERROR is not defined. It is changed to zero and execution proceeds.	None required	Run-time System
ILLEGAL OPERATION - SEQUENTIAL FILE INDEXED FILE WORD-ADDRESSABLE FILE BINARY SEQUENTIAL FILE		5	An attempt has been made to use an incompatible I/O operation on the type of channel specified in the diagnostic (e.g., calling GETLIST on a sequential file).	Self-evident	Run-time System
ILLEGAL STRING INPUT		4	Attempt to read into a string parameter during a call to INPUT or INLIST.	Self-evident	Run-time System

TABLE 15-1. OBJECT TIME DIAGNOSTICS (CONT'D)

Message	Error Key	Chan-Error Key	Significance	Action	Issued By
ILLEGAL STRING OUTPUT		4	Illegal character in string output.	Self-evident	Run-time System
INPUT KIND ERROR		3	Input destination is neither simple variable, array, or subscripted variable.	Self-evident	Run-time System
INSUFFICIENT SCM FOR OVERLAY - OVERLAY INDEX:	5		The indicated overlay requires more SCM/CM.	Increase field length	Run-time System
-I/O ERROR DURING DUMP			Traceback stopped and dump aborted. May be hardware failure.	Notify systems personnel if hardware failure.	Symbolic dump (appears in day-file)
LAYOUT CALL ERROR		6	Procedures established by H END and V END and label set by NODATA are not accessible after return from the layout procedure called by INLIST or OUTLIST.	Self-evident	Run-time System
LCM/ECS ARRAY SIZE ERROR	2		Computed LCM/ECS array size is negative or zero.	Check array bounds.	Run-time System
LCM/ECS STACK OVERFLOW	5		Required array space in LCM/ECS exceeds available memory.	Increase LCM/ECS field length.	Run-time System
LOGARITHM PARAMETER ERROR	4		Argument to LN procedure may not be negative or zero.	Self-evident	Run-time System
NEGATIVE SWITCH INDEX	2		Value of switch designator is negative.	Self-evident	Run-time System

TABLE 15-1. OBJECT TIME DIAGNOSTICS (CONT'D)

Message	Error Key	Chan-Error Key	Significance	Action	Issued By
NON-EXECUTABLE CODE	1		Illegal machine instruction has been encountered.	Verify logic of program; notify systems personnel.	Run-time System
NON-FORMAT INPUT		3	In non-formats I, R, L, or M, input field contains non-octal characters.	Self-evident	Run-time System
NUMBER SYNTAX		3	Number input in standard format does not conform to proper syntax.	Self-evident	Run-time System
NUMERIC INPUT ERROR		3	Data input under format control does not conform in numeric input format.	Self-evident	Run-time System
OPERAND OVERFLOW	1		Numerical value exceeded permissible range.	Self-evident	Run-time System
OPERATION ON ACTIVE CHANNEL		5	An attempt has been made to input or output to an indexed channel while that channel is active.	Only I/O procedures or indexed files are permitted on indexed channel	Run-time System
OUTCHARACTER ERROR		4	Parameter to OUT-CHARACTER call is not in proper range.	Self-evident	Run-time System
OVERLAY FILE NOT PRESENT OVERLAY INDEX:	0		The file for the indicated overlay is not present.	Self-evident	Run-time System
OVERLAY NOT FOUND ON FILE OVERLAY INDEX:	0		The indicated overlay cannot be found on the declared file.	Self-evident	Run-time System
PARAMETER COUNT ERROR	3		Number of actual parameters in procedure call is incorrect.	Self-evident	Run-time System

TABLE 15-1. OBJECT TIME DIAGNOSTICS (CONT'D)

Message	Error Key	Chan. Error Key	Significance	Action	Issued By
PARAMETER KIND ERROR - PARAMETER NO.:	3		Kind of actual parameter in procedure call does not correspond to kind of associated formal parameter.	Self-evident	Run-time System
PARAMETER NOT STRING NOR INTEGER ARRAY	3		A formal I/O string-can only be matched by an actual string or array.	Self-evident	Run-time System
PARAMETER TYPE ERROR PARAMETER NO.	3		Types of actual and formal parameters in procedure call do not correspond.	Self-evident	Run-time System
SCM ARRAY SIZE ERROR	2		Computed SCM array size is negative or zero.	Self-evident	Run-time System
SCM STACK OVERFLOW	5		Data requirements of program exceed available memory.	Increase field length	Run-time System
SIN - COS ERROR	1		Argument to SIN or COS procedure is too large.	Self-evident	Run-time System
SQUARE ROOT PARAMETER ERROR	4		Argument to the SQRT procedure is negative.	Self-evident	Run-time System
STARTING ADDRESS OF MOVE OUT OF ARRAY FIELD - PARAMETER NO:	2		Computed starting address of move is not within array field.	Self-evident	Run-time System
-STOP DUMP TOO LONG			Each call to dump is limited to approximately six pages of printing.	none	Symbolic dump (appears in day file)
STORAGE INCOMPATIBILITY FOR ARRAY	3		Storage allocation (SCM or LCM/ECS) in an actual array differs from that of the format array in a procedure call	Self-evident	Run-time System

TABLE 15-1. OBJECT TIME DIAGNOSTICS (CONT'D)

Message	Error Key	Chan- Error Key	Significance	Action	Issued By
STRING ELEMENT ERROR	6		Rules of STRING ELEMENT violated.	Self-evident	Run-time System
SWITCH BOUNDS ERROR	2		Value of switch designator out of range.	Self-evident	Run-time System
***SYMBOL FILE NOT PROVIDED-OCTAL DUMP REPLACING			Self-evident	Set option D at compile time.	Symbolic dump (appears in dump printout)
SYSPARAMETER ERROR - F -		6	SYSPARAM called with incorrect F parameter.	Self-evident	Run-time System
SYSPARAMETER ERROR - Q -		6	SYSPARAM called with incorrect Q parameter.	Self-evident	Run-time System
TABULATION ERROR		6	Argument of TABULATION not in proper range.	Self-evident	Run-time System
THIS NOT AN ARRAY - PARAMETER NO:	2		The parameter is not an array.	Self-evident	Run-time System
THRESHOLD ERROR	4		Request for invalid function to standard procedure THRESHOLD.	Self-evident	Run-time System
UNASSIGNED CHANNEL		5	No channel defined for channel number used in program.	Supply a channel define statement.	Run-time System
UNCHECKED EOF		2	End-of-partition detected, but no provision made with EOF procedure.	Self-evident	Run-time System
UNCHECKED PARITY		1	No PARITY procedure for detected parity error.	Notify systems personnel.	Run-time System

TABLE 15-1. OBJECT TIME DIAGNOSTICS (CONTD)

Message	Error Key	Chan-Error Key	Significance	Action	Issued By
-UNKNOWN ACTION IN SYMBOL FILE			File given as symbol file in D option has not been created or it has been partially destroyed.	Recompile with correct symbol file.	Symbolic dump (appears in day file)
****WRONG SYMBOL FILE-OCTAL DUMP REPLACING			Symbol file was not created at the last compilation of the program or pertains to another program.	Self-evident	Symbolic dump (appears in dump printout)
ZERO SWITCH INDEX	2		Value of switch designator is zero.	Self-evident	Run-time System

TABLE 15-2. ALGEDIT DIAGNOSTICS

Message	Significance	Action	Issued By
ERR - CH= MUST SPECIFY A TOTAL COUNT LESS THAN 80.	Character count specified exceeds maximum limit.	Reenter command with legal character count.	FORMAT
ERR - COLUMN OR UNIT SPECIFIED BUT NO/TEXT1/.	Column number or unit parameter can be specified only with /text/ parameter.	Reenter command in correct format.	DELETE, LIST, SAVE
ERR - COMPILER NAME REQUIRED.	Compiler name must be specified.	Reenter command with compiler name.	RUN
ERR - EDITFILE EMPTY - NO INFORMATION TO BE ANALYZED.	Self-evident	Load or create file and try again.	ANALYZE
ERR - FILE lfn CANNOT BE FOUND.	File specified is not in list of user's files.	Reenter command with correct file name.	EDIT
ERR - FILE NAME MUST BE ALPHANUMERIC, 7 MAX AND FIRST ALPHABETIC.	File name is specified improperly.	Reenter command with file name in correct format.	EDIT, SAVE
ERR - FILE NAME REQUIRED.	Self-evident	Reenter command with valid file name.	EDIT, SAVE
ERR - FROM Annnnn - INTERNAL ERROR.	Internal error encountered in ALGEDIT routine Annnnn.	Notify Control Data.	
ERR - ILLEGAL COLUMN RANGE.	Column numbers must be in ascending order and column positions must at least equal text string character count.	Reenter command in correct format.	DELETE, LIST, SAVE, TEXT REPLACEMENT
ERR - ILLEGAL LINE RANGE.	Line numbers must be ascending numbers.	Reenter command in correct format.	DELETE, LIST, SAVE, TEXT REPLACEMENT

TABLE 15-2. ALGEDIT DIAGNOSTICS (CONT'D)

Message	Significance	Action	Issued By
ERR - INCREMENT MUST BE GREATER THAN 0.	Self-evident	Reenter command with legal increment value.	ADD, CREATE, RESEQ
ERR - LINE NO. GREATER THAN 999999.	Next line number generated for increment mode or file sequencing exceeds maximum.	Change increment or line value and try again.	ADD, CREATE, EDIT, RESEQ
ERR - LINE NUMBER OUT OF SEQUENCE.	Line numbers are out of order or nonexistent.	Load file using EDIT command with sequence parameter.	EDIT
ERR - LINE NUMBER REQUIRED.	Self-evident	Reenter command with line number.	DELETE
ERR - LINE nnnnnn GREATER THAN FORMAT CH COUNT.	Line nnnnnn expanded beyond maximum character count currently in effect during text replacement operation; truncation did not occur.	No action is required, but truncation will occur when the file is saved (see FORMAT command).	TEXT REPLACEMENT
ERR - LINE nnnnnn TRUNCATED TO 80 CHARS.	Line nnnnnn expanded beyond 80 characters in text replacement operation; truncation occurred.	None	TEXT REPLACEMENT
ERR - LINES GREATER THAN 80 CHARS. WERE TRUNCATED.	Lines in EDIT file in excess of 80 characters were truncated. Files with lines greater than 80 characters cannot be edited.	None	EDIT
ERR - LINES TRAUNCATED: CH=nn CHARS LONGEST LINE WAS mm.	Lines in excess of maximum of nn were truncated; longest line was mm characters.	If truncation is not acceptable, change character count and try again.	RUN, SAVE
ERR - NO INFORMATION IN EDIT FILE.	Self-evident	Load or create file and try again.	ADD, LIST, DELETE, RESEQ, RUN, SAVE, TEXT REPLACEMENT

TABLE 15-2. ALGEDIT DIAGNOSTICS (CONT'D)

Message	Significance	Action	Issued By
ERR - ON OR OFF REQUIRED.	Self-evident	Reenter command with ON or OFF.	RECHECK, CHECK
ERR - OVERWRITE ILLEGAL ON PERM FILE.	Specified file already exists as attached permanent file.	Return or discard permanent file and try again, or save user file under different name. If message is displayed after RUN command, notify system personnel.	RUN, SAVE
ERR - PARAM n: COLUMN SPECIFICATION INCOMPLETE.	Column numbers in parameters are not specified correctly: (col-1, col-2).	Reenter command in correct format.	DELETE, LIST, TEXT REPLACEMENT
ERR - PARAM n: DUPLICATE PARAMETER	Parameter n appears twice in command.	Reenter command in correct format.	ADD, CREATE, DELETE, EDIT, FORMAT, LIST, RECHECK, RESEQ, RUN, SAVE, TEXT REPLACEMENT, ANALYZE, CHECK
ERR - PARAM n: ILLEGAL LINE NUMBER	Line number parameter outside legal range.	Reenter command in correct format.	ADD, CREATE, RESEQ, TEXT REPLACEMENT
ERR - PARAM n: NUMERIC PARAMETER REQUIRED.	Self-evident	Reenter command in correct format.	ADD, CREATE, FORMAT, RESEQ
ERR - PARAM n: TOO MANY DIGITS.	Parameter n exceeds maximum range. Line numbers and increment values are 1-6 digits; column numbers and tab positions are 1-2 digits, LTBL value is 1-4 digits.	Reenter command in correct format.	ADD, CREATE, DELETE, FORMAT, LIST, RESEQ, SAVE, LINE=TEXT, TEXT REPLACEMENT, ANALYZE

TABLE 15-2. ALGEDIT DIAGNOSTICS (CONT'D)

Message	Significance	Action	Issued By
ERR - PARAM n: UNRECOGNIZABLE PARAMETER.	Self-evident	Check legal delimiters, separators, and spelling. Reenter command in correct format.	ADD, BYE, CREATE, DELETE, EDIT, FORMAT, LIST, RECHECK, RESEQ, RUN, SAVE, TEXT REPLACEMENT, ANALYZE, CHECK
ERR - PARAM n: /TEXT2/ REQUIRED.	Text replacement string is required following equals sign.	Reenter command in correct form (null text replacement string is specified as //.)	TEXT REPLACEMENT
ERR - RESERVED FILE NAME.	User has specified reserved file name.	Reenter command specifying nonreserved user file name.	EDIT, RUN, SAVE
ERR - TAB= MUST SPECIFY ONLY 1 CHAR.	Self-evident	Check that tab character parameter is followed by legal separator. Reenter command.	FORMAT
ERR - TABLE LENGTH MUST BE GREATER THAN ZERO.	Self-evident	Reenter command with positive LTBL value.	ANALYZE, CHECK
ERR - TABS TOO BIG OR OUT OF ORDER.	Tab positions must be in ascending order and cannot exceed 80 characters.	Reenter command in correct format.	FORMAT
ERR - TOO MANY PARAMETERS.	Self-evident	Reenter command in correct format with fewer parameters.	ADD, CREATE, FORMAT, RESEQ, RUN
ERR - TOO MANY TABS.	User's tab settings exceed installation maximum limit.	Reenter command with fewer tab settings or request installation to extend limit.	FORMAT
ERR - /TEXT1/ MUST HAVE AT LEAST 1 CHAR.	Text search string cannot be null.	Reenter command in correct format.	DELETE, LIST, SAVE, TEXT REPLACEMENT
ERR - UNRECOGNIZABLE COMMAND.	Check legal delimiters, separators, and spelling.	Reenter command in correct form.	

TABLE 15-2. ALGEDIT DIAGNOSTICS (CONT'D)

Message	Significance	Action	Issued By
ERR - lfn ALREADY EXISTS.	User already has a local file by this name.	Discard, drop, or rename permanent file, or specify overwrite. Then, reenter command in correct format.	SAVE
ERR - lfn CONNECTED TO TERMINAL.	File is currently connected to user's terminal.	Disconnect file and reenter command in correct format.	EDIT, RUN, SAVE
ERR - lfn IMPROPERLY ATTACHED FOR THIS OPERATION.	File is permanent and does not have permission required for this operation.	Attach file with required passwords; reenter command in correct format.	EDIT, RUN, SAVE
FROM ALGEDIT.	An internal error has occurred.	Notify Control Data.	
WRNG - ADD WONT REPLACE OR BYPASS LINES	Add will not bypass or replace existing lines.	Reenter command with new line and increment values.	ADD
WRNG - AN ABORT PROCESSING WAS PERFORMED - GO ON.	An interrupt has been processed.	Continue	interrupt
WRNG - CHECKING MODE ALREADY ON - GO ON.	The user was already in checking mode.	Continue	CHECK
WRNG - CHECKING MODE ALREADY OFF - GO ON	The user was not in checking mode.	Continue	CHECK
WRNG - EDIT FILE NOT SAVED.	Entered command would destroy user's EDIT file; command was not executed.	Save or delete edit file, or reenter command to delete file automatically.	BYE, CREATE, EDIT
WRNG - NO SUCH LINES.	No lines satisfy the specified command requirements.	Reenter command in correct format.	LIST, DELETE, SAVE, TEXT REPLACEMENT

TABLE 15-2. ALGEDIT DIAGNOSTICS (CONT'D)

Message	Significance	Action	Issued By
WRNG - TABS GREATER THAN CH VALUE CLEARED.	User specified new character count; tab settings that were cleared from previous format specification contain column number beyond new count.	None.	FORMAT
WRNG - TEXT TRUNCATED TO 20 CHARS - AUTOMATIC VETO SET.	User specified text string greater than 20 characters; automatic VETO option has been set.	None.	LIST, DELETE, SAVE, TEXT REPLACEMENT
WRNG - xxxxxx TRUNCATED FROM LONG LINE.	Line exceeds terminal's current maximum character count; first 7 characters truncated were xxxxxxx.	Reenter command in correct format or change format specifications.	ADD, CREATE, LINE=TEXT

The ALGOL Interactive Debugging Aids (AIDA) constitute an interactive debugging facility for the ALGOL 4 compiler.

AIDA allows the user to control program execution through the use of breakpoints. Execution is suspended and control is given to the user when a breakpoint is executed. The user can check intermediate values, make decisions, enter or release breakpoints, and then resume execution of his program until the next breakpoint is reached.

16.1 INTERACTIVE MODE

AIDA normally executes in interactive mode. The automatic field length reduction initiated by the operating system must be inhibited to execute an ALGOL program. For programs run under INTERCOM under NOS/BE, enter REDUCE,OFF. For programs run under NOS, enter REDUCE(-). Programs run under SCOPE 2 cannot utilize AIDA in interactive mode.

16.1.1 COMPILATION AND EXECUTION

If AIDA is used, a program must be compiled and executed with the interactive (Q) option to produce an interactive file. If no file name was specified by the Q option at compile time (see chapter 6), the interactive file name used is QFILE. Otherwise, it is the file name specified. This same file must be specified by the Q option at execution time (see chapter 8); the default is QFILE.

If the Q option is not used both at compile time and at execution time or if the same interactive file is not used at both times, a fatal error occurs.

16.1.2 INTERACTIVE FILES

AIDA uses five standard files for interactive execution (besides QFILE, if used as the interactive file):

INALG is the standard input file (channel 60).

OUTALG is the standard output file (channel 61).

INPUT is the terminal input file.

OUTPUT is the terminal output file.

HISTORY contains a copy of the files INPUT and OUTPUT. User entries are preceded by the symbols / /.

16.1.3 INITIAL DIALOG

The initial breakpoint is entered into the beginning of the ALGOL program by AIDA after declarations for the outermost block have been processed. When this breakpoint is executed, the following message appears:

header line

date time

HAVE YOU USED QFILE BEFORE

The user should answer NO if this program has not been executed previously with this interactive file. If the user answers YES, the following message appears:

WILL YOU KEEP PREVIOUS BREAKPOINTS

If the user answers YES, the breakpoints established within the interactive file the last time it was used are retained. If the user answers NO, all breakpoints are released, the interactive file is initialized, and the following message appears:

AIDA - - - INITIAL BREAKPOINT

PLEASE TYPE A COMMAND

At this point any valid command may be entered. These commands are discussed in section 16.2.

16.1.4 DISPLAY FORMATS

The symbol ../.. is displayed on the last line of every page to signify that a new page follows. On terminals that display one full page at a time, it is necessary to press the carriage return to see the next page; a heading appears at the top of every page identifying the line at which the program is currently stopped and the action in progress. On terminals that display lines continuously, with no page break, this heading is not displayed, and the new page appears immediately after the old page.

Messages generated by AIDA appear in either a long or a short format, at the user's option.

Examples:

LONG	SHORT
NEXT COMMAND PLEASE	++
BREAKPOINTS RELEASED FROM 5 TO 15	OFF FROM 5 TO 15
AIDA - - - FULL BREAK AT LINE 7	FULL, LINE 7
PARTIAL BREAK SET AT LINE 9	PARTIAL B AT 9

AIDA generates messages initially in long format. The following command can be used at any time to change the current format to the alternate format:

MESSAGE (or M)

16.2 INTERACTIVE COMMANDS

The following commands may be executed when the program has stopped execution at a breakpoint.

Keyword	Abbreviation	Function	Section
ABORT	A	Aborts the program	16.2.7
BREAK	B	Sets breakpoints	16.2.4
CONIN	CIN	Connects INALG	16.2.9
CONOUT	COT	Connects OUTALG	16.2.10
DISOUT	DOT	Disconnects OUTALG	16.2.10
END	E	Ends execution normally	16.2.7
GO	G	Resumes execution	16.2.7
MESSAGE	M	Changes message format	16.1.4
NO	N	Response to AIDA questions	16.2.3
RELEASE	R	Releases breakpoints	16.2.5
STOP	S	Ends an AIDA display	16.2.3
TEACH	T	Lists AIDA commands	16.2.2
YES	Y	Response to AIDA questions	16.2.3
=	=	Displays variables	16.2.6
:=	:=	Modifies variables	16.2.6
*	*	Comments	16.2.8

16.2.1 COMMAND SYNTAX

An AIDA command normally starts with a keyword (= and := are exceptions). Some keywords stand alone; others are followed by a variable number of parameters separated by commas. If the command is terminated by a semicolon, information after the semicolon is considered a comment. No terminator is needed if no comment follows the command. Embedded blanks within the command are ignored. Keywords may be abbreviated.

If a command is spelled incorrectly, the following message is displayed:

- WRONG COMMAND
- TRY AGAIN

If the syntax of a command is incorrect, the following message is displayed:

- SYNTAX ERROR IN COMMAND
- input string up to the wrong character
- TRY AGAIN

If the answer is still incorrect, the following message is displayed:

- SYNTAX ERROR IN ANSWER
- TRY AGAIN

A syntactic description of the AIDA commands follows:

$\langle \text{symbol} \rangle ::= \langle \text{letter} \rangle | \langle \text{digit} \rangle | \langle \text{logical} \rangle | \langle \text{delimiter} \rangle$
 $\langle \text{empty} \rangle ::= \langle \text{the null string of symbols} \rangle$
 $\langle \text{letter} \rangle ::= a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z$
 $\langle \text{digit} \rangle ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9$
 $\langle \text{logical} \rangle ::= \text{true} | \text{false}$
 $\langle \text{bracket} \rangle ::= (|) | [|]$
 $\langle \text{separator} \rangle ::= , | . | _ | 10 | := | = | / | + | - | * | ;$
 $\langle \text{delimiter} \rangle ::= \langle \text{bracket} \rangle | \langle \text{separator} \rangle$
 $\langle \text{ident} \rangle ::= \langle \text{letter} \rangle | \langle \text{ident} \rangle \langle \text{letter} \rangle | \langle \text{ident} \rangle \langle \text{digit} \rangle$
 $\langle \text{unsintgr} \rangle ::= \langle \text{digit} \rangle | \langle \text{unsintgr} \rangle \langle \text{digit} \rangle$
 $\langle \text{integer} \rangle ::= \langle \text{unsintgr} \rangle | + \langle \text{unsintgr} \rangle | - \langle \text{unsintgr} \rangle$
 $\langle \text{fraction} \rangle ::= . \langle \text{unsintgr} \rangle$
 $\langle \text{expon} \rangle ::= 10 \langle \text{integer} \rangle$
 $\langle \text{decimal} \rangle ::= \langle \text{unsintgr} \rangle | \langle \text{fraction} \rangle | \langle \text{unsintgr} \rangle \langle \text{fraction} \rangle$
 $\langle \text{unsnumb} \rangle ::= \langle \text{decimal} \rangle | \langle \text{expon} \rangle | \langle \text{decimal} \rangle \langle \text{expon} \rangle$
 $\langle \text{number} \rangle ::= \langle \text{unsnumb} \rangle | + \langle \text{unsnumb} \rangle | - \langle \text{unsnumb} \rangle$
 $\langle \text{subscr} \rangle ::= \langle \text{integer} \rangle | *$
 $\langle \text{subslst} \rangle ::= \langle \text{subscr} \rangle | \langle \text{subslst} \rangle , \langle \text{subscr} \rangle$
 $\langle \text{subsvar} \rangle ::= \langle \text{ident} \rangle [\langle \text{subslst} \rangle]$
 $\langle \text{variable} \rangle ::= \langle \text{ident} \rangle | \langle \text{subsvar} \rangle$
 $\langle \text{display} \rangle ::= \langle \text{variable} \rangle =$
 $\langle \text{assign} \rangle ::= \langle \text{variable} \rangle : = \langle \text{number} \rangle | \langle \text{variable} \rangle : = \langle \text{logical} \rangle$
 $\langle \text{break} \rangle ::= \text{BREAK} | \text{B}$
 $\langle \text{singline} \rangle ::= \langle \text{number} \rangle | \langle \text{singline} \rangle , \langle \text{number} \rangle$
 $\langle \text{rangline} \rangle ::= \langle \text{number} \rangle / \langle \text{number} \rangle | \langle \text{rangline} \rangle , \langle \text{number} \rangle / \langle \text{number} \rangle$
 $\langle \text{lines} \rangle ::= \langle \text{singline} \rangle | \langle \text{rangline} \rangle$
 $\langle \text{num} \rangle ::= \langle \text{number} \rangle | \langle \text{empty} \rangle$

```

<limits>::=[ <num> , <num> , <num> ] | <empty>
<varlist>::=<ident> | <varlist> , <ident>
<lists>::=( <varlist> ) | <empty>
<brkpoint>::=<break> , <lines> <limits> <lists>
<release>::=release | r
<clrbreak>::=<release> , <lines>
<go>::=go | g
<continue>::=<go>
<abort>::=abort | a
<conin>::=conin | cin
<conout>::=conout | cot
<disout>::=disout | dot
<end>::=end | e
<message>::=message | m
<teach>::=teach | t
<no>::=no | n
<stop>::=stop | s
<yes>::=yes | y
<text>::=<any character or characters available on the CRT or TTY> | <empty>
<special>::=<abort> | <conin> | <conout> | <disout> | <end> | <message> | <teach> | * <text>
<dialog>::=<no> | <stop> | <yes> | ☐
<command>::=<display> | <assign> | <continue> | <brkpoint> | <clrbreak> | <special> | <dialog> |
    <command>; <text>

```

16.2.2 GETTING HELP FROM AIDA

A list of the valid AIDA commands, including a brief description of each, may be obtained at execution time with the command:

TEACH (or T)

16.2.3 RESPONDING TO AIDA

AIDA requires a response in two situations. In the first case, AIDA will ask a question. Either of the following responses are valid:

YES (or Y)

NO (or N)

In the second case, the message PAUSE will appear at the bottom of a display page. If a space followed by a carriage return is entered, AIDA will continue to display output. The display can be ended with the command:

STOP (or S)

If the display is ended, AIDA will display:

NEXT COMMAND PLEASE

and any valid AIDA command may be entered.

16.2.4 SETTING BREAKPOINTS

Breakpoints may be set with either form of the command:

BREAK, l_1, l_2, \dots, l_n [i, f, s] (v_1, \dots, v_5) (or B)

BREAK, $l_1/l_2, \dots, l_m/l_n$ [i, f, s] (v_1, \dots, v_5)

where:

l_1, l_2, \dots, l_n are the source line numbers where breakpoints are to be set.

$l_1/l_2, \dots, l_m/l_n$ are ranges of source line numbers where breakpoints are to be set.

The two forms of the command may not be mixed; single source line numbers and ranges of source line numbers must be specified in separate commands.

i, f, and s are decimal numbers representing INITIAL, FINAL, and STEP. The values determine what happens when the breakpoint is executed. INITIAL is the first iteration at which the program stops, FINAL is the last iteration, and STEP is the increment used each time the breakpoint is executed. All three are optional. If none are specified, the brackets may be omitted. If the brackets are used, the commas must be included whether or not all the parameters are specified. The default value for all three parameters is 1. The values of i, f, and s should satisfy the following conditions:

$$\begin{aligned} i &\leq f \\ i, f, s &\leq 2^{20} - 1 \\ i, f, s &\geq 1 \end{aligned}$$

If these conditions are not satisfied, the breakpoint status is set to FULL (see below).

v_1, \dots, v_5 are variables to be displayed when the breakpoint is executed. Neither subscripted variables nor arrays are allowed in this list. No variables need be supplied, and only the first five variables listed are used; a warning message is displayed if more than five variables are listed. A warning message is also displayed if a variable has not been declared. Excess variables or variables not declared are ignored.

Examples:

BREAK, 15, 38, 45

B, 50(A, B, C, D, E)

B, 45 [5, 13, 3] (I, A, B); INITIAL = 5, FINAL = 13, STEP = 3.

Examples of warning messages:

BREAK, 145 (I, J, K, L, M, N, O, P)

ONLY FIRST FIVE CORRECT VARIABLES ACCEPTED

BREAK, 20(A, B, C)

A = NOT DECLARED

A breakpoint is defined at each line specified in the command if possible. Breakpoints are set at each line of a range if possible. Breakpoints are not set at any of the following lines:

Lines in the middle of declarations or specification.

Lines in the middle of expressions (an expression is always terminated by an ALGOL keyword or delimiter).

Lines in the middle of a for clause (between for and do).

Lines in the middle of an actual parameter list.

Comment lines before the first begin or after the last end.

If an unbreakable line is specified, the breakpoint is set at the next breakable line. If a line beyond the last source line of the program is specified, no breakpoint is set and the following message is displayed:

LAST BREAKABLE LINE IS n

Example:

In this example, breakable lines are indicated by an asterisk in the left margin.

```
PROGRAM TO SHOW BREAKABLE LINES
  begin integer I,J;
  boolean B; integer
* array A [1:10];
* B:=true;
* comment ASSIGN VALUES TO ARRAY;
  for I:=0 step 1
  until 9
* do
  begin J:=I
* +1;
  if B then A[J]:=I
* else
* A[J] :=-I;
  OUTPUT(61,'-3ZD,/',
* A[J]);
* if B then
  B:=false
* else
  B:=
  true
* end OF LOOP;
  end OF DEMONSTRATION PROGRAM
```

A breakpoint is always executed after the line in which it was set. If the statement causes control to be transferred to another part of the program (such as a go to statement), the breakpoint is not executed.

The status of a breakpoint, established when it is set by a BREAK command, determines the action that occurs when it is executed:

The status of a breakpoint is set to full if INITIAL, FINAL, and STEP are not specified or do not satisfy the conditions specified for them above. At a full breakpoint, execution stops, values of variables are displayed if previously requested, and any valid AIDA command can be entered.

The status of a breakpoint is set to partial if INITIAL, FINAL, or STEP are specified and they satisfy the conditions specified for them above. The action at a partial breakpoint depends on the value of the internal counter associated with this breakpoint. When the value of this counter is equal to INITIAL or FINAL, execution stops, values of variables are displayed, if requested previously, and any valid AIDA command can be entered. When the value of this counter is greater than INITIAL and less than FINAL, AIDA reports that the breakpoint has been encountered and values of variables are displayed if previously requested, but execution does not stop.

When variables or arrays are displayed at a full breakpoint or any type of partial breakpoint, the program pauses approximately every 18 lines with the message PAUSE at the bottom right-hand corner of the display screen. In response, the user may either type a space (followed by the send-key) to continue display or else type the command STOP to end the display. When display is so terminated, AIDA responds with the message NEXT COMMAND PLEASE and any valid AIDA command may be entered.

Each time a partial breakpoint is executed, its internal counter is incremented by STEP. The breakpoint is ignored if its internal counter is outside the range defined by INITIAL and FINAL.

16.2.5 RELEASING BREAKPOINTS

Breakpoints are released by either form of the following command:

RELEASE, l_1, l_2, \dots, l_n (R may be entered instead of RELEASE)

RELEASE, $l_1/l_2, \dots, l_m/l_n$

where:

l_1, l_2, \dots, l_n are the source line numbers of the breakpoints to be released.

$l_1/l_2, \dots, l_m/l_n$ are ranges of source line numbers of breakpoints to be released.

The two forms of the command may not be mixed.

16.2.6 DISPLAYING AND MODIFYING VARIABLES

The values of variables may be displayed with the command:

variable name =

The values of variables may be modified with the command:

variable name :=

Only the first ten characters of the variable name are used; additional characters are ignored. Only simple and subscripted variables and arrays within the appropriate scope (as defined in chapter 2, section 2.7) can be displayed or modified; neither expressions nor variables are allowed as subscripts. Formal parameters that have been specified by value can be displayed or modified, but not those called by name. No switches, labels, strings, or function procedures can be displayed or modified.

When a variable is modified, the new value must agree with the type (Boolean, real, or integer).

Examples:

r =

b := false

AIDA uses the following formats to display values, depending on the variable type:

integer	7ZD if the value has 8 digits or less D.7D ¹⁰ +3D if the value has more than 8 digits
real	D.7D ¹⁰ +3D
Boolean	*T* or *F*

If a requested variable is not accessible, not declared, or not an allowed type, an informative message is displayed, and the command is ignored.

Examples:

W=NOT ACTIVE

Z=LABEL

Y=STRING

PAR=NOT BY VALUE

P=FUNCTION OR PROCEDURE

S=SWITCH

A=NOT DECLARED

XYZ=INTEGER

2 DIMENSIONS DECLARED

An entire array may be displayed or modified by entering its name without subscripts. Rows or columns of an array also may be displayed or modified by replacing the appropriate subscripts with asterisks.

Examples:

An array is declared as follows:

```
integer array ABC[1 : 3, 1 : 3]
```

The following commands could be entered:

```
ABC:=5;           the entire array is set to 5
ABC[*,1]:=10;     is the same as for I:=1 step 1 until 3 do ABC[I,1]:=10;
```

After execution of these two commands, the command:

```
ABC=
```

would display:

```
ABC[1,1]=10
ABC[1,2]=5
ABC[1,3]=5
ABC[2,1]=10
ABC[2,2]=5
ABC[2,3]=5
ABC[3,1]=10
ABC[3,2]=5
ABC[3,3]=5
```

16.2.7 RESTARTING, TERMINATING, AND ABORTING

When program execution stops at a breakpoint, this command may be used to resume execution:

```
GO      (or G)
```

Execution continues until the next breakpoint is executed or until the program ends normally if no further breakpoint is encountered.

Execution can be terminated with either of these commands:

```
END      (or E)
```

```
ABORT    (or A)
```

Both END and ABORT commands cause a warning message to be displayed. A response is required — YES to terminate execution and leave AIDA, or NO to continue interactive execution through AIDA.

The END command terminates execution normally at that point. All files are closed and final messages are displayed.

The ABORT command terminates execution abnormally. A traceback dump is triggered, all files are closed, and final messages are displayed. The format of the traceback dump is determined by the D option (see section 8.2).

When a run is ended, whether through normal termination or through execution of the END command or the ABORT command, files INPUT and OUTPUT are always disconnected. Files INALG and OUTALG are disconnected only if they have been previously connected.

16.2.8 COMMENTS

Comments may be entered with:

*comment

Comments have no effect on program execution; they are listed in the HISTORY file only. Any character from the character set is acceptable in a comment.

Any information following a semicolon in any AIDA command is considered to be a comment.

16.2.9 CONNECTING THE INPUT FILE

Data for a program may be entered directly from the terminal by connecting the file INALG with the command:

CONIN (or CIN)

A warning message is displayed before INALG is connected. A response is required — yes to connect INALG or no to leave INALG disconnected. Once INALG has been connected, it cannot be disconnected.

When INALG is connected, there is no indication when the program is waiting for input unless that indication is provided by the program. Any data entered while INALG is connected must conform to the format given in the corresponding input statement.

16.2.10 CONNECTING THE OUTPUT FILE

Output from the program may be displayed at the terminal by connecting the file OUTALG with the command:

CONOUT (or COT)

The file OUTALG may be disconnected with the command:

DISOUT (or DOT)

Once OUTALG has been connected, the system abort commands are needed to end output prematurely; AIDA does not have control while program output is being displayed.

16.3 FATAL ERRORS

When a fatal error occurs, AIDA displays an error message and aborts. Fatal errors can be caused by incorrect use of the interactive file (QFILE) or by an internal compile error.

The following error messages might be displayed:

QFILE DOES NOT MATCH LGO

ERROR ON FILE QFILE

ERROR ON FILE OUTPUT OR HISTORY

16.4 USING AIDA IN BATCH MODE

Although AIDA is intended to be used interactively at a terminal, it is possible to use AIDA in batch mode. In this case, a record containing all AIDA commands must be supplied in the input file.

OUTALG and HISTORY are not printed automatically at the end of the job. If copies of these files are desired, control statements must be added to the job to rewind them and copy them to a print file.

The commands CONIN and CONOUT have no meaning in batch mode and should not be used. Results are unpredictable if they are used.

Example:

Job statement.
Accounting statements
ALGOL,Q.
COPYBR,INPUT,INALG.
REWIND,INALG.
RFL(4500)
REDUCE(-) (NOS only)
LGO,Q.
REWIND,OUTPUT,HISTORY.
COPY,OUTALG,OUTPUT.
COPY,HISTORY,OUTPUT.
7/8/9 card
Source Program
7/8/9 card
Input Data
7/8/9 card
Commands for AIDA
7/8/9 card
6/7/8/9 card

The following is an example of interactive use of AIDA under INTERCOM. Underlining denotes the user's response to INTERCOM or AIDA. Delimiters for the ALGOL basic symbols appear as " rather than ≠ because the terminal used for this example uses the ASCII character set.

• • / • •

• • / • •

A I D A - - - INITIAL BREAKPOINT

PLEASE TYPE A COMMAND BREAK,7 ← Request breakpoint set at line 7.
 FULL BREAK SET AT LINE 19 ← Line 7 is unbreakable; breakpoint set at next
 NEXT COMMAND PLEASE GO ← breakable line.
 /.. ← Resume execution.

A I D A - - - FULL BREAK AT LINE 19

NEXT COMMAND PLEASE CONOUT ← Connect file OUTALG for display of program
 FILE OUTALG IS NOW CONNECTED output at terminal.
 NEXT COMMAND PLEASE GO

1	1	1.00000	} ← Output of program.
2	1	2.00000	
3	2	1.50000	
4	3	1.66667	
5	5		

 /..

ALGOL 4.1, AIDA LEVEL 2
 YOU NOW HAVE A RECORD OF THIS RUN IN FILE HISTORY
 YOU MAY OBTAIN IT BY DISPOSE OR BATCH
 OR PAGE IT ON YOUR TERMINAL
 +++END OF INTERACTIVE ALGOL RUN+++

END OF ALGOL 4.1 LEVEL 0002
 .314 CP SECONDS EXECUTION TIME
 COMMAND- REWIND,LGO. ← Rewind LGO for reexecution.
 COMMAND- LGO,Q.

CHANNEL,-2=HISTORY,P136,PP60
 CHANNEL,-1=OUTPUT,P30,PP130000
 CHANNEL,60=INALG,P30
 CHANNEL,61=OUTALG,P136,PP60
 /..

ALGOL 4.1, AIDA LEVEL 2
 10/07/74 10.50.54.
 HAVE YOU USED QFILE BEFORE YES ← QFILE has been previously used when executing
 this program.
 WILL YOU KEEP PREVIOUS BREAKPOINTS YES ← Retain breakpoints from last execution with AIDA.
 /..

A I D A - - - INITIAL BREAKPOINT

PLEASE TYPE A COMMAND * SECOND RUN ← Enter comment in file HISTORY.
 NEXT COMMAND PLEASE DISOUT; DISCONNECT FOR THIS RUN ←
 Disconnect file OUTALG so that program output
 will not be displayed at terminal. Comment fol-
 lowing semicolon entered in file HISTORY.

OUTALG HAS BEEN DISCONNECTED
 NEXT COMMAND PLEASE BREAK,6 ← Set full breakpoint at line 6.

FULL BREAK SET AT LINE 6
 NEXT COMMAND PLEASE GO

.../...

A I D A - - - FULL BREAK AT LINE 6

NEXT COMMAND PLEASE N := 10 ← User makes typographical error.

SYNTAX ERROR IN COMMAND
 N

TRY AGAIN. N := 10 ← Set value of N to 10.

N = 10
 NEXT COMMAND PLEASE GO

.../...

A I D A - - - FULL BREAK AT LINE 19 ← Breakpoint at line 19 set during previous execution.

NEXT COMMAND PLEASE BREAK,30[6,8](N,I) ← User forgets comma following FINAL specification when omitting STEP specification.

SYNTAX ERROR IN COMMAND

BREAK,30[6,8]

TRY AGAIN BREAK,30[6,8,](N,I) ← Set partial breakpoint at line 30 to display values of N and I during last three iterations of loop.

PARTIAL BREAK SET AT LINE 30
 NEXT COMMAND PLEASE GO

.../...

STOPPED AT INITIAL AT LINE 30
 COUNTER 6

N = 10
 I = 8

NEXT COMMAND PLEASE PHI[8] = ← Display value of PHI[8], which has just been computed.

PHI[8] = 1.6153846**+000

NEXT COMMAND PLEASE GO

A I D A - - - PARTIAL AT 30 COUNTER 7

N = 10
 I = 9

} AIDA does not stop at partial breakpoint when COUNTER ≠ INITIAL or FINAL.
 .../...

STOPPED AT FINAL AT LINE 30
 COUNTER 8

N = 10
 I = 10

NEXT COMMAND PLEASE PHI[10] = ← Display value of PHI[10], which has just been computed.

PHI[10] = 1.6176471**+000

NEXT COMMAND PLEASE BREAK,33 ← Set full breakpoint at line 33.

FULL BREAK SET AT LINE 33
 NEXT COMMAND PLEASE GO

.../...

A I D A - - - FULL BREAK AT LINE 33

NEXT COMMAND PLEASE

FIBONAC[* ,2] = ; LIST FIBONACCI NUMBERS

- - - - ARRAY FIBONAC

[1,2] = 1
[2,2] = 1
[3,2] = 2
[4,2] = 3
[5,2] = 5
[6,2] = 8
[7,2] = 13
[8,2] = 21
[9,2] = 34
[10,2] = 55

NEXT COMMAND PLEASE

GO

←
List second column of array FIBONAC, which
contains the first ten Fibonacci numbers.

.../...

ALGOL 4.1, AIDA LEVEL 2
YOU NOW HAVE A RECORD OF THIS RUN IN FILE HISTORY
YOU MAY OBTAIN IT BY DISPOSE OR BATCH
OR PAGE IT ON YOUR TERMINAL
+++END OF INTERACTIVE ALGOL RUN+++

END OF ALGOL 4.1 LEVEL 0002
1.397 CP SECONDS EXECUTION TIME
COMMAND-

17.1 ALGOL INTERACTIVE SYNTAX CHECKER

The ALGOL Interactive Syntax Checker (ALGEDIT) is available for interactive execution under TELEX for NOS and INTERCOM for NOS/BE . Users executing under SCOPE 2 cannot use ALGEDIT. ALGEDIT provides the capability for text editing and for checking ALGOL text either line by line or by block without the need for a full compilation. Syntactic and lexicographic diagnostics generated by ALGEDIT refer to line numbers entered at the terminal, so that the errors can be easily corrected. The user can turn syntax analysis on or off at any time.

Using ALGEDIT, the programmer can create, examine, and modify coded sequential files and check syntax from a remote terminal. Each ALGEDIT user is assigned a scratch file called the edit file. Coded sequential files to which the user has access can be copied into the edit file for subsequent editing. Any file created or modified in the edit file can be saved on mass storage or submitted directly to the operating system for compilation and execution by the ALGOL, COBOL 4, or FORTRAN Extended (FTN) compiler, or by the COMPASS assembler.

The basic unit of information in ALGEDIT is a line of text, which can vary in length up to 80 characters. A line number, either entered by the user or generated by ALGEDIT, is appended to each text line. The last line entered in or displayed from the edit file is by definition the current line.

Text line formats are predefined for ALGOL, COMPASS, COBOL 4 and FORTRAN Extended programming languages; however, the user can alter tabulation and margin controls or define a new format.

User files can be listed, deleted, and modified. The list, delete, and modify operations can be performed on a single text line, on a range of text lines (up to a complete file), or on character strings within text lines.

The two modes of operation are command mode, initiated when ALGEDIT is called by the user, and text mode, initiated when the user enters commands to create or insert text lines with line numbers automatically generated by ALGEDIT.

Operating system control statements can be entered during operation under ALGEDIT. However, if a control statement (including a program call statement) is used as a command and has the same name as an ALGEDIT command, it must be terminated by a period or right parenthesis. If the operating system control statement name is unique, it can be entered without the terminator, which is subsequently added by ALGEDIT.

INTERCOM commands (including ETL and EFL), NOS interactive commands, and NOS control statements prefixed by a label, /, or \$, cannot be entered under ALGEDIT.

If comments are appended to control statements, the command text must be terminated by a period.

All commands and text lines are entered by pressing the return key on the teletype or the send key on the display keyboard.

17.2 ALGEDIT COMMAND SYNTAX

An ALGEDIT command consists of a command verb that can be followed by parameters. Some verbs require no parameters; others require at least one. A command verb with associated parameters must be entered as one line.

The command formats described in this section are intended to guide the programmer in using ALGEDIT commands. The following notational conventions are used:

Brackets [] indicate terms that can be included or omitted as desired by the user.

When terms are enclosed between braces { }, one item must be chosen; the others are to be omitted.

An ellipsis ... immediately following a statement element indicates it can be repeated at the user's option.

Special characters (table 17-1) are essential where shown.

All words shown in lowercase letters represent information that the user is to supply. These words generally indicate the nature of the information they represent (file name, line or column number, and so forth).

All words printed entirely in uppercase letters have preassigned meanings to ALGEDIT; they include command verbs and keywords.

Command verbs and keywords can be abbreviated to a unique set of characters; the minimum number of characters required is underlined. Additional characters can be specified up to the complete verb or keyword, but character sequences must be correct.

The command verb must appear at the beginning of the command statement. Most command parameters are position-independent and can appear in any order; exceptions are noted in the command descriptions.

All examples in this part are shown as displayed on a teletypewriter; data entered by the user is underscored.

17.3 SPECIAL CHARACTERS

Characters that have special functions in ALGEDIT are described in table 17-1. Most of them have special meaning only in ALGEDIT commands; elsewhere they are recognized as valid data characters. Others can be entered only for specific functions and cannot be used as data characters.

17.4 ACCESSING ALGEDIT

ALGEDIT commands can be entered in one of two modes:

ALGEDIT Command Mode

ALGEDIT Text Mode

Table 17-1. Special Characters

Characters	Function
=	Used as a separator in ALGEDIT commands. In text replacement it must be specified between the text strings; within commands it must be used to separate keywords and associated variables. When text lines are entered singly, the equals sign must be specified immediately after the line number. In text mode, when a line number other than the one displayed is to be entered, an equals sign must both precede and follow the new line number. When entered as the only character, an equals sign can also be used to terminate text mode. In most other situations, the equals sign can be used as a valid data character.
Blanks or commas	Either blanks or commas are used as parameter separators; they are equivalent. Adjacent blanks and commas are interpreted as a single separator. Both can be used as valid data characters; however, trailing blanks are truncated on input to the edit file.
Interrupt characters	When an interrupt command is entered, current action is suspended or terminated. Control of a terminal under INTERCOM is returned to ALGEDIT command mode; control of a terminal under NOS 1 is returned to NOS 1 interactive command mode. If an interrupt command is entered while ALGEDIT is deleting, resequencing, or replacing text lines, the edit file is left in an unknown condition.
Text string delimiters / or any character other than blank, comma, parenthesis, equals sign, letter, or digit	Delimits text character strings on input; the symbol selected must appear as the first and last character of the string. If the delimiter character is used as a data character within a string, it must be specified twice or a different character must be used to delimit the string. For example, A/B as a text character string within a command can be entered as /A/B/, or as *A/B*.
()	Parentheses are used to delimit column numbers in ALGEDIT commands. Elsewhere they can be used as valid data characters.
*EOR	This character string is entered, beginning in column 1, each time an end of section is required in a user's file. When the file is saved on mass storage, an end of section is written in that position. Conversely, when a file is read, each end of section is inserted in the edit file as *EOR.
*EOF	This character string is equivalent to an end of partition. The operating system forces an end of section before an end of partition. If a user inserts *EOF into the edit file without a preceding *EOR, an *EOR will precede *EOF after a save/edit sequence; therefore, the file should be sequenced to allow generation of a line number for the *EOR.

17.4.1 ALGEDIT COMMAND MODE

To call ALGEDIT, the users enters:

ALGEDIT.

ALGEDIT indicates readiness to receive input by displaying two consecutive periods:

• •

If ALGEDIT is being used under NOS 1, the two periods are replaced by a question mark:

?

In ALGEDIT command mode (indicated by the two periods or question mark) certain NOS 1 and NOS/BE 1 control statements can be entered as well as ALGEDIT commands.

ALGEDIT responds to a command of the form:

line=text

with only a line feed under INTERCOM and a question mark under NOS 1. ALGEDIT responds to all other valid commands, including operating system control statements, with two consecutive periods or a question mark. At this point, another command can be entered.

Although certain NOS 1 or NOS/BE 1 commands can be entered in ALGEDIT command mode, execution of these commands is not as efficient as when they are entered in NOS 1 interactive or INTERCOM command mode; response time might be affected adversely. If many operating system commands are to be entered in succession, the user should leave ALGEDIT.

17.4.2 ALGEDIT TEXT MODE

Entering a CREATE or ADD command initiates text mode. In this mode of operation, lines of text can be entered in the edit file. Line numbers automatically generated by ALGEDIT are displayed at the terminal.

In ALGEDIT text mode, a line number and an equals sign are displayed. The user can then enter the text to be associated with that line number. ALGEDIT generates a new line number (the previous number incremented by a defined value) and displays the new number and an equals sign.

This process continues until text mode is terminated by either the user or ALGEDIT. The ALGEDIT text mode is terminated when:

The user is inserting new text lines between existing lines in the edit file and the value of a generated line number equals or exceeds the next existing line number.

A single equals sign or an interrupt command is entered.

17.5 EXITING FROM ALGEDIT

To leave ALGEDIT, the user enters:

BYE or END

To preclude inadvertent destruction of the edit file, the message:

WRNG-EDIT FILES NOT SAVED

is displayed by ALGEDIT if the edit file has not been saved since it was last modified. The user can either save the edit file and reenter the BYE command, or reenter the BYE command if the file is not to be saved. Control returns to system command mode.

If the contents of the edit file are not to be saved, ALGEDIT can be exited immediately through the command:

BYE, BYE

Control is returned to system command mode, and the edit file is no longer available. The command END, END is not allowed.

Examples:

To leave ALGEDIT after saving the edit file:

. . BYE

User enters BYE.

Control returns to system command mode.

To leave ALGEDIT before saving a modified edit file:

. . B

User enters BYE.

WRNG-EDIT FILE NOT SAVED

System displays a warning message.

. . SAVE ABC

User enters SAVE,filename.

. . B

User reenters BYE.

Control returns to system command mode.

To leave ALGEDIT without saving the edit file:

. . B,B

User enters BYE,BYE.

Control returns to system command mode.

17.6 FILE CREATION AND MODIFICATION

ALGEDIT provides extensive text editing capabilities:

New files can be created or existing files modified.

Program formatting can be automatic or declared by the user.

Lines can be listed, deleted, added, searched, and replaced.

Text character strings can be replaced within lines.

Line numbers can be automatically generated and displayed by ALGEDIT or entered by the user.

Line numbers within existing files can be resequenced.

17.6.1 FORMAT COMMAND

In ALGEDIT command mode, the ALGOL format specification is in effect by default. The tabular column positions, valid tabulating character, and maximum character count per input line are controlled by this specification. The **FORMAT** command can be used to establish other formats, either predefined or supplied by the user. Specifications established with the command remain in effect for the duration of the user's session with ALGEDIT or until changed by the user. The command can also be used to obtain a list of format specifications currently available at the terminal.

Every line entered in the edit file from the terminal keyboard is governed by the format specification, which dictates the maximum character count and columnar positions for each input line. In addition, every character entered is checked against the tabulating character. When a valid tabulating character is encountered, blanks are inserted in the data line from that point up to the tabular position where the next data character is placed. The blank fill is an internal process; spaces do not appear on the terminal display at the time the line is entered. If a tabulating character is entered when no tabular positions exist, the tabulating character is accepted as a valid data character.

If lines entered from the terminal exceed the specified character count, they are truncated to the maximum allowed, and a message is displayed at the terminal.

To change or list the format specifications, the user enters:

FORMAT $\left[\begin{array}{l} \text{, format-name} \\ \left[\text{, TAB=char} \right] \left[\text{, tab-1} \left[\text{, tab-2} \left[\dots \left[\text{, tab-n} \right] \right] \right] \right] \left[\text{, CH=nn} \right] \\ \text{, SHOW} \end{array} \right]$

format-name

Established format for data lines entered from the terminal. The format-name must be one of the following:

ALGOL

COBOL

COMPASS

FORTRAN

<u>TAB</u> =char	Keyword. char is any valid character on the terminal keyboard (except % on a 200 User Terminal). The character specified becomes the tab character checked on input.
tab-1, tab-2, . . . , tab-n	Column position: 1 or 2 digits in the range 0 through 80. Tab column numbers (up to 10) must be specified in ascending order.
<u>CH</u> =nn	Keyword. CH establishes the maximum character count for each input line. nn is the maximum character count, 1 or 2 digits in the range 1 through 80. This count is also checked when either a SAVE or RUN command is entered.
<u>SHOW</u>	Keyword. The current format specification is listed at the terminal in the form:

CH=nn TAB CHAR=char TAB COL=t1, t2, . . . , tn

where nn is the maximum character count, char is the tab character, and t is the tab column position (10 positions maximum).

One, two, or all tabulation parameters, including character count, can be entered in one FORMAT command (the tab column positions are interpreted by ALGEDIT as one parameter). Any omitted parameter remains unchanged from the current value.

If the FORMAT command is entered with no parameters, the tab column position is set to zero, the maximum character count is set to accept lines up to 80 characters, and no tab character check is made.

When a format name is entered in the FORMAT command, a format specification is established at the terminal that enables the user to enter lines in the format of a specific language, as listed below:

ALGOL:

CHARACTER COUNT = 72

TAB CHARACTER = \$

TAB COLUMNS = 7 10 13 16 19

COBOL:

CHARACTER COUNT = 72

TAB CHARACTER = ;

TAB COLUMNS = 8 12 16 20 24

COMPASS:

CHARACTER COUNT = 72

TAB CHARACTER = ;

TAB COLUMNS = 11 18 36

FORTTRAN:

CHARACTER COUNT = 72

TAB CHARACTER = ;

TAB COLUMNS = 7

Examples:

To list the format specification currently in effect at the terminal:

. . <u>FORM S</u>	User enters FORMAT, SHOW.
CH= 72 TAB CHAR=\$ TAB COL=7 10 13 16 19	System lists the current specifications.
. .	ALGEDIT is ready for the next command.

To define a user format where the colon is the tab character, the maximum character count is 50 and the tab columns are 10, 20, 30, and 40:

. . <u>F 10, 20, 30, 40 C=50 TAB=:</u>	User enters FORMAT, t1, t2, t3, t4, CH=nn,
	TAB=char.
. .	ALGEDIT is ready for the next command.

To change only the tab character in the current format specification, setting the tab character to a down arrow:

. . <u>FOR T=↓</u>	User enters FORMAT, TAB=char.
. .	ALGEDIT is ready for the next command.

17.6.2 RECHECK COMMAND

The user enters the RECHECK command to echo each line entered at the terminal:

RECHECK, { ON }
 { OFF }

ON Keyword. Turns on rechecking mode.

OFF Keyword. Turns off rechecking mode.

17.6.3 CREATE COMMAND

The CREATE command is used to construct a new file:

CREATE [, line [, incr]]

line	The first line number to be displayed at the terminal: 1 to 6 digits from 1 to 999999. If line is omitted, the system assumes default (100 in the released system) first line number.
incr	The value to be used for line number incrementation after each text entry: 1 to 6 digits from 1 to 999999. If incr is omitted, the system assumes default (10 in the released system) increment value.

When the CREATE command is accepted, text mode is initiated; the first line number and an equals sign are displayed at the terminal. The user can enter a text line of 1 to 80 characters; the maximum line length depends on the current format specification. At least one blank character (space) must be entered to produce a blank text line; merely pressing the carriage return does not create a new line.

Text mode remains in effect until the user terminates with an interrupt command or returns to ALGEDIT command mode by entering a single equals sign (table 17-1).

While in text mode, the user can enter a line number other than that displayed by entering a command of the following form:

=line=text

line is a number of 1 to 6 digits, and text is a text line of 1 to 80 characters, whose length depends on the current format specification. After this line is entered, text mode resumes at the point of interrupt.

If the CREATE command is entered when the user has an existing edit file that has not been saved since it was last modified, ALGEDIT ignores the command and displays the message:

WRNG-EDIT FILE NOT SAVED

The user can enter the SAVE command and then reenter the CREATE command, or if the edit file is not to be saved, reenter only the CREATE command. In the latter case, contents of the edit file are destroyed.

Examples:

To create a file of two lines with a first line number of 10 and an increment value of 10:

. . <u>CREATE 10 10</u>	User enters CREATE, line, incr.
10= <u>FIRST LINE</u>	User enters text line 1.
20= <u>SECOND LINE</u>	User enters text line 2.
30= <u>=</u>	User enters an equals sign to interrupt text mode.
. .	ALGEDIT is ready for the next command.

To create a file of three lines using default line and increment values, and to correct a mistake in the first line:

..	<u>CR</u>	User enters CREATE.
	100= <u> #BGIN# </u>	User enters the first line.
	110= =100= <u> #BEGIN# </u>	User corrects the first line.
	110= <u> OUTPUT(61, #(#(#PROGRAM#)#)#)# </u> ;	ALGEDIT reissues the line number.
	120= <u> #END# </u>	User enters the third line.
	130= <u> = </u>	User enters an equals sign to interrupt text mode.
..		ALGEDIT is ready for the next command.

17.6.4 ADD COMMAND

The ADD command is used to insert new lines between existing lines or to add lines at the end of the edit file. The ADD command cannot be used to replace or bypass existing lines unless the OVERWRITE parameter is specified.

ADD [, line [, incr]] [, OVERWRITE]

line Line number: 1 to 6 digits, from 1 to 999999. The first line number displayed at the terminal. If line is omitted, the system assumes the last line number in the edit file plus the default increment value (10 in the released system).

incr Increment value: 1 to 6 digits, from 1 to 999999. If incr is omitted, the system assumes the default increment value.

OVERWRITE Keyword. OVERWRITE permits the user to replace or bypass existing line numbers.

Text mode is initiated by the ADD command, and the first line number and an equals sign are displayed at the terminal. A 1- to 80-character line of text, whose maximum length depends on the current format specification, can be entered. At least one blank character (space) must be entered to produce a blank or zero-length text line.

Text mode remains in effect until it is terminated by the user with an interrupt command, a single equals sign (table 17-1) is entered, or an existing line number is encountered (unless OVERWRITE is specified). The system then returns to ALGEDIT command mode.

In text mode, a command of the following form can be used to enter any line number other than that displayed.

=line=text

line is the desired line number of 1 to 6 digits, and text is a line of text of 1 to 80 characters where length depends on the current format specification.

Examples:

To add new text lines between lines 10 and 20 in the edit file and correct an error in the first line entered:

. . <u>AD 13 3</u>	User enters ADD, line, incr.
13= <u>INSET ONE</u>	User enters the first line.
16= <u>=13= INSERT ONE</u>	User corrects the spelling in the first line.
16= <u>INSERT TWO</u>	System redisplay the line number.
19= <u>INSERT THREE</u>	
WRNG-ADD WONT REPLACE OR BYPASS LINES	System displays an error message; the next line would exceed line 20 in the edit file.
. .	ALGEDIT is ready for the next command.

To add a line to the end of the edit file (last existing line number is 300, default increment value is 10):

. . <u>AD</u>	User enters ADD.
310= <u>\$\$=END#;</u>	User enters text.
320= <u>=</u>	User enters an equals sign to terminate text mode.
. .	ALGEDIT is ready for the next command.

17.6.5 LINE=TEXT COMMAND

To place one line of data in the edit file while in either ALGEDIT command mode or text mode, the user can enter the following form:

[=] line=text

line	Line number: 1 to 6 digits, from 1 to 999999. In text mode, an equals sign must precede the line.
------	---

text	Line of text: 0 to 80 characters. The maximum line length depends on the character count established by the format specification.
------	---

This command does not affect the terminal's mode of operation. The entered line can replace an existing line or insert a new line in the edit file. If no text is entered (the text line is zero length), a blank line appears at the specified line number of the edit file.

ALGEDIT issues only a line feed (not two periods) in response to a valid line=text command; then any command can be entered.

In text mode, the command must be entered in the form =line=text: the last line number displayed at the terminal is displayed again to allow the next text line to be entered in proper sequence.

Examples:

To enter line 352 in the edit file with the text, "this is line 352":

. . <u>352=THIS IS LINE 352</u>	User enters the line in the form: line=text
	ALGEDIT responds with a line feed only.

To correct a line in text mode:

. . <u>CRE</u>	User enters CREATE.
100= \$ MINE ≠BEGIN≠	Text mode is initiated; user enters the text.
200= \$ ≠ARRAY≠A[100:200];	System displays the next line number; user enters the text.
300= =100=\$ MINE ≠BEGIN≠ ≠INTEGER≠1;	User enters an equals sign, the number of the line to be corrected, and the new text for line 100.
300= \$\$INPUT(60,≠(≠≠)≠,I);	System displays the last line number previously displayed; user enters the text.
400= =	User enters an equals sign to terminate text mode.
. . .	ALGEDIT is ready for the next command.

17.6.6 DELETE COMMAND

The DELETE command is used to delete lines from the edit file:

$$\underline{\text{DELETE}}, \left\{ \begin{array}{l} \underline{\text{ALL}} \\ \text{line-1} \\ \underline{\text{LAST}} \end{array} \right\} \left[\begin{array}{l} \text{line-2} \\ \underline{\text{LAST}} \end{array} \right] \left[\begin{array}{l} \text{, /text/} \\ \left[\begin{array}{l} \text{(col-1} \\ \text{[, col-2)]} \end{array} \right] \end{array} \right] \left[\begin{array}{l} \text{[, UNIT]} \\ \text{[, VETO]} \end{array} \right]$$

<u>ALL</u>	Keyword. All lines in the edit file are deleted or searched for the text search string.
line-1	Line number: 1 to 6 digits, from 1 to 999999. The line is the first or only line to be deleted or searched for.
line-2	Line number: 1 to 6 digits, from 2 to 999999. The line is the last line to be deleted or searched for in a range beginning at line-1.
<u>LAST</u>	Keyword. As the first parameter, LAST causes the last line in the edit file to be deleted or searched. As the second parameter, LAST causes the deletion or search of the lines, beginning at line-1 and extending through the last line in the file.
/text/	Text search string: 1 to 20 characters delimited by slashes or an equivalent delimiter. The file is searched for this string (the search can be restricted to a range of line and column numbers). Lines containing this string are deleted from the edit file. (See table 17-1, text string delimiters.)
col-1	Column number: 1 or 2 digits, from 1 to 80. col-1 is the first or only column number of the text string search. It must be preceded by a left parenthesis and followed by either a comma and col-2, or by a right parenthesis.
col-2	Column number: 1 or 2 digits, from 2 to 80. col-2 is the last column number to be searched in a range beginning at col-1. It must be greater in value than col-1 and followed by a right parenthesis. The range must be greater than or equal to the number of characters specified in the text search string. Column specification is significant only if a text search string is specified. Lines are deleted only if the text string occurs within the range, or when a single column is specified and the string begins in col-1.

UNIT Keyword. UNIT dictates that the text search string must appear as a unit within a line; that is, the text string must be delimited by characters (including blank, beginning-of-line, and end-of-line), other than letters or digits. All other symbols are permissible delimiters. Delimiters need not be identical; for example, a string can be preceded by a blank and followed by a slash.

VETO Keyword. VETO permits the user to approve deletions before they occur. The line to be deleted is displayed at the terminal. The user can enter:

YES to delete the line.

CONTINUE to delete the line and any subsequent lines that satisfy the requirements specified in the DELETE command.

Any other response retains the line.

The DELETE command must include at least one parameter: ALL, LAST, or a line number.

If a text search string is specified in the command, a message reports the number of deletions performed:

n DELETIONS

where n is the number of lines deleted.

If more than 20 characters are entered as a text search string, the string is truncated to the first 20 characters. An informative message is displayed at the terminal and VETO automatically takes effect. Each line that satisfies the search conditions is displayed. The user can enter the VETO responses given above.

An interrupt command can be entered to terminate execution of a DELETE command; however, the edit file might be left with the specified lines partially deleted.

Examples:

To delete line 100 in the edit file:

. . DELETE 100
. .

User enters DELETE, line-1
ALGEDIT is ready for the next command.

To delete from lines 200 through the last line only if the character string AX appears within columns 7 through 72:

. . D,/AX/, (7, 72), 200, L
2 DELETIONS
. .

User enters DELETE,/text/, (col-1, col-2), line-1,
LAST.
System issues a message.
ALGEDIT is ready for the next command.

To delete all lines from line 100 through line 200:

. . DE 100 200
. .

User enters DELETE, line-1, line-2.
ALGEDIT is ready for the next command.

To delete all lines in the edit file so that a new file can be constructed:

. . <u>DEL AL</u>	User enters DELETE, ALL
. .	ALGEDIT is ready for the next command.

To delete all lines in the edit file, with veto power, only if they contain the character C in column 1 as a unit:

. . <u>D, A, /C/, (1), U, V</u>	User enters DELETE, ALL, /text/, (col-1), UNIT, VETO.
20=C BEGIN DO LOOP	System displays the qualifying line.
<u>N</u>	User elects to retain the line.
50=C END SCAN	System displays the qualifying line.
<u>N</u>	User elects to retain the line.
0 DELETIONS	System issues a message; all qualifying lines have been displayed, none were deleted.
. .	ALGEDIT is ready for the next command.

17.6.7 RESEQ COMMAND

The RESEQ command is used to resequence the line numbers in the edit file:

RESEQ [, line [, incr]]

line	Line number at which resequencing is to begin: 1 to 6 digits, from 1 to 999999. If line is omitted, the system assumes the default value (100 in the released system).
incr	Increment value: 1 to 6 digits, from 1 to 999999. If incr is omitted, the system assumes a default increment value (10 in the released system).

On acceptance of this command, ALGEDIT resequences all line numbers in the edit file. New line numbers are written over existing line numbers, and the current line number pointer is reset to the first line number in the file.

A RESEQ command can be terminated by an interrupt command; however, the result might be a partially resequenced edit file that should be resequenced before further editing. In some cases, unpredictable results can occur.

Examples:

To resequence the edit file with a first line number of 100 and increment value of 100:

. . <u>RES 100 100</u>	User enters RESEQ, line, incr.
. .	ALGEDIT is ready for the next command.

To resequence with the default first line number and increment value:

. . RESEQ
 . .

User enters RESEQ.
 ALGEDIT is ready for the next command.

17.6.8 TEXT REPLACEMENT COMMAND

The text replacement command is used to replace text strings in lines of the edit file:

$$/text-1/= /text-2/ \left[\left\{ \begin{array}{c} \underline{ALL} \\ line-1 \\ \underline{LAST} \end{array} \right\} \left[\left\{ \begin{array}{c} line-2 \\ \underline{LAST} \end{array} \right\} \right] \right] \left[, (col-1 [, col-2]) \right] [, \underline{UNIT}] [, \underline{VETO}]$$

/text-1/= /text-2/ Text strings. The equals sign must be specified with no spaces on either side.

/text-1/ Text search string: 1 to 20 characters delimited by slashes or an equivalent delimiter. The file is searched for this string; the search can be restricted to a range of line and column numbers. (See text string delimiters, table 17-1.)

/text-2/ Text replacement string: 0 to 20 characters delimited by slashes or an equivalent delimiter. The replacement string replaces the text search string when conditions of the search are satisfied. (See text string delimiters, table 17-1.)

ALL Keyword. ALL causes a search of all lines in the edit file.

line-1 Line number. 1 to 6 digits, from 1 to 999999. Line-1 is the first or only line to be searched.

line-2 Line number: 1 to 6 digits, from 2 to 999999. Line-2 is the last line to be searched in a range beginning at line-1.

LAST Keyword. As first parameter, LAST causes a search of the last line in the file; as the second parameter, LAST causes a search beginning at line-1 and extending through the last line in the file.

col-1 Column number: 1 or 2 digits, from 1 to 80; the first or only column to be searched. col-1 must be preceded by a left parenthesis and followed by either a comma and col-2 or by a right parenthesis.

col-2 Column number: 1 or 2 digits, from 2 to 80; the last column to be searched in a range beginning at col-1. col-2 must be greater than col-1 and must be followed by a right parenthesis. The range must be greater than or equal to the number of characters in the text search string.

Replacement takes place only if the text string occurs within the column range, or (when a single column is specified) if the text string begins in col-1.

UNIT Keyword. UNIT dictates that the text search string must appear as a unit within a line. The text string must be delimited by characters other than letters or digits (including the blank, end-of-line, and beginning-of-line).

VETO Keyword. VETO permits the user to approve text replacement before it occurs. The changed form of the line is displayed at the terminal. The user can enter:

YES to accept the change.

CONTINUE to accept the change and any subsequent changes that satisfy the requirements specified in the text replacement command.

Any other response retains the original line.

If the text replacement command is entered with no parameters, the search is performed on the line to which the current line pointer is set.

The number of replacements performed is reported in a message:

n CHANGES

n is the number of changes made. Because more than one replacement can occur in any line, the number of changes displayed might differ from the number of lines changed.

The two text strings specified as the command verb need not contain the same number of characters; the line affected is expanded or contracted as necessary. If the maximum character count is exceeded, the replacement occurs, and an informative message is displayed. Truncation occurs if a line exceeds 80 characters.

The text replacement string can be entered as a null string (two consecutive slashes, no embedded blanks). This specification causes the text search string to be deleted if all search conditions are satisfied.

When more than 20 characters are entered as a text search or text replacement string, the string is truncated to the first 20 characters. An informative message is displayed at the terminal and VETO automatically takes effect. The changed form of each line that satisfies the search conditions is displayed. The user can enter the VETO responses given above.

The text replacement command cannot be used to edit a tabulation character (as defined by the FORMAT command) into an existing line. Such an entry is accepted as a data character; no tabulation occurs.

Examples:

To replace the variable name AX with the name BZ in the current line, where AX must be a unit:

. ./AX/=BZ/,U
1 CHANGES
..

User enters /text-1/=text-2/, UNIT
System issues a message.
ALGEDIT is ready for the next command.

To replace the character strings TCS with the string TERMINAL CONTROL SYSTEM whenever TCS appears in the edit file (two text replacement commands must be entered because the text replacement string is greater than 20 characters); the user also requests VETO power:

<pre>. . /TCS=/TERMINAL CONTROL SY./,A,V 60=THE TERMINAL CONTROL SY. HAS THE YES 190=IN THE TERMINAL CONTROL SY. USERS MAY Y 2000=* * *TERMINAL CONTROL SY. ABORT * * * N 2 CHANGES . . /SY./=SYSTEM/ 60 190 U 2 CHANGES . .</pre>	<pre>User enters /text-1=/text-2/, ALL, VETO System displays the changed line. User accepts the change. System displays the changed line. User accepts the change. System displays the changed line. User retains the original line. System issues a message. User enters /text-1=/text-2/, line-1, line-2, UNIT System issues a message. ALGEDIT is ready for the next command.</pre>
--	--

To replace the character C with the character * only if C appears as a unit in column 1 (all lines are searched):

<pre>. . /C=/*/ A (1) U 15 CHANGES . .</pre>	<pre>User enters /text-1=/text-2/, ALL,(col-1), UNIT System issues a message. ALGEDIT is ready for the next command.</pre>
--	--

To replace the character string PROGRAM in line 2310 with a null string (line currently appears as 2310=END PROGRAM), the user requests VETO power:

<pre>. . /PROGRAM=/, 2310, V. 2310=END YES 1 CHANGES . .</pre>	<pre>User enters /text-1=/text-2/, line-1, VETO System displays the changed line. User accepts the change. System issues a message. ALGEDIT is ready for the next command.</pre>
--	--

17.7 FILE STORAGE AND DISPLAY

Three ALGEDIT commands can be used to store and display files:

EDIT	Loads local files into edit file.
SAVE	Saves edit file as local file.
LIST	Displays edit file or selected file portions at terminal.

17.7.1 EDIT COMMAND

The EDIT command is used to load a local file into the edit file:

EDIT, filename [, SEQUENCE]

filename	Name of file to be edited. The name is required immediately following the command verb.
<u>SEQUENCE</u>	Keyword. ALGEDIT line numbers are assigned to each line as they are entered into the edit file. If omitted, the system assumes that ALGEDIT line numbers already exist in the local file.

The file named can be any coded sequential file to which the user has read access, including local and attached permanent files. The file to be loaded is called the source file; it is not modified by execution of the EDIT command, except that it might be repositioned.

If the user enters the EDIT command when the existing edit file has not been saved since it was last modified, ALGEDIT ignores the command and displays the message:

WRNG-EDIT FILE NOT SAVED

The existing file can be saved by the SAVE command, or if the contents of the edit file need not be retained, the EDIT command should be reentered. In the latter case, the contents of the current edit file are destroyed.

When loading a file created outside of ALGEDIT, the user is required to sequence the file with line numbers. When SEQUENCE is specified, ALGEDIT line numbers beginning with the installation defined first line number are appended to each line of the file. The source file is not affected; line numbers appear only in the edit file. Consequently, the length of each line in the edit file is increased by six characters. Because lines in the edit file are restricted to a maximum of 80 characters, truncation can occur; if it does, an informative message is displayed.

Multisection files can be loaded for editing, but they appear in the edit file as one section. When an end of section is encountered in the source file, the character string *EOR is assigned a sequential line number and written in the edit file to indicate an end of section condition. For end of partition, the character string *EOF is generated.

Example:

To load the local file named AFIL into the edit file:

. . <u>E AFIL</u>	User enters EDIT, filename
. .	ALGEDIT is ready for the next command

To load the local file BFIL into the edit file with line number sequencing:

. . <u>ED,BFIL S</u>	User enters EDIT, filename, SEQUENCE
. .	ALGEDIT is ready for the next command.

Examples:

To save the edit file under the file name ALGPRG:

. . <u>SA ALGPRG</u>	User enters SAVE, filename
. .	ALGEDIT is ready for the next command.

To save the edit file in place of an existing local file named ALGDATA, with a line length of 30 characters and no ALGEDIT line numbers:

. . <u>F CH=30</u>	User enters FORMAT, CH=nn
. . <u>SA, ALGDATA, O, N</u>	User enters SAVE, filename, OVERWRITE, NOSEQ
. .	ALGEDIT is ready for the next command.

To save all lines between line 10 and line 100 that have the character * in column 1:

. . <u>SA LEFEN 10, 100 /*/ (1)</u>	User enters SAVE, filename, line-1, line-2, /text/, col-1
. .	ALGEDIT is ready for the next command.

To merge a portion of file MAINA and all of files SUB1 and SUB2 into a single file named COMBO that contains a main program and two subroutines:

. . <u>ED, MAINA</u>	User enters EDIT, filename
. . <u>SA, COMBO, M, 100, L, N</u>	User enters SAVE, filename, MERGE, line-1, LAST, NOSEQ
. . <u>E, SUB1</u>	User enters EDIT, filename
. . <u>SA, COMBO, N, M</u>	User enters SAVE, filename, NOSEQ, MERGE
. . <u>E, SUB2</u>	User enters EDIT, filename
. . <u>SA, COMBO, M, N</u>	User enters SAVE, filename, MERGE, NOSEQ
. .	ALGEDIT is ready for the next command.

The merged file COMBO can be loaded into the edit file with line number sequencing for examination or editing, and then saved as a local file by using the OVERWRITE parameter.

. . <u>E, COMBO, S</u>	User enters EDIT, filename, SEQUENCE
. .	User modifies file COMBO
. . <u>SA, COMBO, O</u>	User enters SAVE, filename, OVERWRITE
. .	ALGEDIT is ready for the next command.

17.7.3 LIST COMMAND

The LIST command enables the user to display all or selected portions of the EDIT file at the terminal.

$$\text{LIST} \left[\left\{ \begin{array}{c} \text{ALL} \\ \text{line-1} \\ \text{LAST} \end{array} \right\} \left[\left\{ \begin{array}{c} \text{line-2} \\ \text{LAST} \end{array} \right\} \right] \right] \left[\text{,/text/} \left[\text{(col-1 [, col-2])} \right] \left[\text{, UNIT} \right] \right]$$

<u>ALL</u>	Keyword. All lines in the file are listed or searched for the text search string.
line-1	Line number: 1 to 6 digits, from 1 to 999999. The line is the first or only line to be listed or searched.
line-2	Line number: 1 to 6 digits, from 2 to 999999. The line is the last line to be listed or searched in a range beginning at line-1.
<u>LAST</u>	Keyword. If LAST is specified as the first parameter, the last line in the file is displayed or searched; if LAST is specified as the second parameter, the listing or search begins at line-1 and continues through the last line in the file.
/text/	Text search string: 1 to 20 characters delimited by slashes or an equivalent delimiter. The file is searched for this string (the search can be restricted to a range of line and column numbers). Lines containing the text string are listed at the terminal. (See text string delimiters, table 17-1.)
col-1	Column number: 1 or 2 digits, from 1 to 80. col-1 is the first or only column number of a text string search. It must be preceded by a left parenthesis and followed by either col-2 or a right parenthesis.
col-2	Column number: 1 or 2 digits, 2 to 80. col-2 is the last column number of a text string search in a range beginning in col-1. It must be greater in value than col-1 and followed by a right parenthesis. The range of columns must be greater than or equal to the number of characters in the text string. Column specification is significant only if a text search string is specified. Lines are listed only if the text string occurs within the range, or if it begins in col-1 when only a single column is specified.
<u>UNIT</u>	Keyword. UNIT dictates that the text search string must appear as a unit within a line, that is, the text string must be delimited by characters (including blank, beginning-of-line, and end-of-line) other than letters or digits. All other symbols are permissible delimiters. Delimiters need not be identical: for example, a string can be preceded by a blank and followed by a slash.

If the LIST command is entered with no parameters, the current line (line to which the edit file pointer is set) is listed. The current line is the last line displayed, inserted, or deleted, or — if a resequencing has occurred or a new file has been loaded in the edit file — the first line of the file.

Lines that satisfy the LIST command requirements are displayed in the form:

line number=text line

Examples:

To list lines 10 through 20 of the edit file, and the current line:

<pre> . . <u>L</u> 10 20 10= DATA 10 15= DATA 15 20= DATA 20 . . <u>L</u> 20= DATA 20 . . </pre>	<p>User enters LIST, line-1, line-2 System lists the appropriate lines.</p> <p>User enters LIST System lists the current line. ALGEDIT is ready for next command</p>
--	--

To list all lines in the edit file that contain the variable AX as a unit:

<pre> . . <u>LI, A/AX/, U</u> 115= ≠REAL≠AX, AY, AZ; 140= AX:=X↑2; 620= OUTPUT(61,≠(≠≠)≠, AX); . . </pre>	<p>User enters LIST, ALL, /text/, UNIT System lists all the lines that satisfy command requirements.</p> <p>ALGEDIT is ready for the next command.</p>
---	--

17.8 SYNTAX CHECKING

ALGEDIT includes a facility for checking the syntax of an ALGOL program, either line-by-line or on an entire block. When an error is detected, an appropriate message is displayed on the terminal beside the number of the erroneous line in the edit file. A message of the form:

RECOVERY: xxxxxxxx REPLACED BY yyyyyyy

indicates that ALGEDIT has diagnosed xxxxxxxx as syntactically incorrect and, to agree with previous lines, has replaced it with yyyyyyy. The line in the edit file remains as it was originally entered, however, and the user should correct it using the commands described in section 17.6.

Syntax checking detects an error only after the first correctly spelled ≠BEGIN≠. For example, if the edit file contains:

```

100= ≠BEHAN≠
110= ≠INTEGER≠ I;
120= I:=1;
130= ≠END≠

```

no errors are detected by the commands CHECK, ON and ANALYZE, because the four lines are considered part of <pre> (chapter 4).

Syntax checking always begins with the first line of the file, and ends with the last line, or with the first *EOR or *EOF encountered.

17.8.1 CHECK COMMAND

Entering a CHECK command initiates or terminates line-by-line syntax checking during creation of an ALGOL program in the edit file:

CHECK, $\left\{ \begin{array}{l} \underline{\text{ON}} \\ \underline{\text{OFF}} \end{array} \right\}$

ON Turns on the checking mode.

OFF Turns off the checking mode.

Checking mode initiated by the command CHECK, ON remains in effect until the user enters a RUN, EDIT, or CHECK, OFF command or a NOS 1, or NOS/BE 1 control statement.

In checking mode, the syntax of each line is checked as it is entered by the user. Syntactic errors and inconsistencies with lines previously entered are reported in messages displayed on the terminal. This checking is performed when the user enters an insertion, deletion, or text replacement command, as well as an addition to the end of the edit file.

Example:

```
ALGEDIT
. .
. . CH ON
. .
CR
100=
≠BEGIN≠≠INTEGER≠A, B, C;
110=
≠PROCEDURE≠P(O);≠REAL≠O;
120=
≠BEGIN≠INTEGER≠Z;
=>=>=> LINE 120                      RECOVERY: INTEGER≠Z;
REPLACED BY ≠INTEGER≠Z;
130=
=120=≠BEGIN≠≠INTEGER≠Z;
130=
.
.
.
```

In response to the error message generated by ALGEDIT, the user has corrected line 120 by entering

= 120 = ... line ...

instead of the text for line 130. Automatic incrementing causes the system to respond 130 = again.

17.8.2 ANALYZE COMMAND

Entering ANALYZE causes the syntax of all lines in the edit file to be checked:

ANALYZE [LTBL=xxxxxx]

xxxxxx Field length reserved for the symbol table used by ALGEDIT for the syntax analysis.

Before entering ANALYZE, the user should add `≠EOP≠` following the last line in the edit file to indicate to ALGEDIT that the user considers the blocks in the edit file to be complete. Otherwise, the ALGEDIT syntax analyzer does not compare the number of `≠BEGIN≠` statements with the number of `≠END≠` statements. The `≠EOP≠` following the last program in the edit file should be deleted before execution; otherwise, the compiler generates a fatal diagnostic reporting an empty input file.

The optional suffix to the ANALYZE command should be used to increase the field length reserved for the symbol table when a large program is to be analyzed. The standard field length allocated to this table is 1000 (octal). If the user enters a smaller value on the ANALYZE command, the standard field length remains in effect.

Example:

EDIT, XFILE

LIST, ALL

```
10 = ≠BEGIN≠ ≠BOOLEAN≠F; ≠INTEGER≠I, N;
20 = F:= ≠TRUE≠; N:=3;
30 = L1: ≠BEGIN≠
40 = ≠BEGIN≠
50 = ≠OWN≠ ≠REAL≠ ≠ARRAY≠A[1:3];
60 = OUTPUT (61, ≠ (≠ ≠) ≠, A); ≠END≠;
70 = ≠BEGIN≠ ≠ARRAY≠B [1:N];
80 = ≠FOR≠ I:=1 ≠STEP≠ 1 ≠UNTIL≠ N-1 ≠DO≠
90 = B[I] :=I;
100 = OUTPUT (61, ≠ (≠ ≠) ≠, B);
110 = ≠IF≠ F ≠THEN≠ ≠BEGIN≠
120 = F:= ≠FALSE≠; ≠GOTO≠ L1
130 =                      ≠END≠;
140 = ≠END≠;
150 = ≠EOP≠
```

ANALYZE

```
150 = ≠EOP≠
=>=>=> LINE 150                      RECOVERY: 2 ≠END≠ (S)
MISSING... ADDED BY COMPILER
```

Example:

AN, L=10000

Now, new length of symbol table (TBL) is 10000

AN, LTBL=100

Standard length of TBL is unchanged, 1000 remains effective.

17.9 COMPILATION AND EXECUTION

ALGEDIT allows the user to transfer files to another available system at appropriate times, and to interrupt a program during its execution.

17.9.1 RUN COMMAND

With the RUN command, a local file or a program in the edit file can be transferred to one of the available systems for assembly or compilation and execution.

RUN[,system-name [,FILE=filename] [,NOEX][,SUP]

system-name Compiler or assembler name. The name must be one of the following:

<u>ALGOL</u>	ALGOL compiler (default)
<u>COBOL</u>	COBOL 4 compiler
<u>COMPASS</u>	COMPASS assembler
<u>FTN</u>	FORTTRAN Extended (FTN) compiler

FILE=filename File name. The name of the local file to be transferred. (Embedded blanks are not allowed.) If the name is omitted, contents of the edit file are transferred for compilation.

NOEX Keyword. NOEX inhibits execution. If it is omitted, execution is performed.

SUP Keyword. Under NOS SUP is ignored; fatal, nonfatal, and informative diagnostics are always displayed at the terminal. Under INTERCOM, if SUP is omitted, a call to ERRORS is executed causing display of error messages to the terminal. SUP suppresses the call to ERRORS, and only fatal errors are displayed.

In response to the RUN command, ALGEDIT creates (or rewinds and overwrites) the file LGO and connects the files INPUT and OUTPUT to the user's terminal just before program execution. The file LGO contains the binary code generated by the assembler or compiler; this file can be executed later in the terminal session without reassembly or recompilation of the source program. If an ALGOL program is to be executed, the user should inhibit automatic field length reduction prior to entering ALGEDIT. This can be done under INTERCOM by entering the command REDUCE, OFF, and under NOS by entering the command REDUCE(-).

The file INPUT is used for terminal entry of data expected by the executing program; when an ALGOL program is to be executed, this file is the standard ALGOL input file, channel 60. The file OUTPUT is used to display output from the executing program; when an ALGOL program is to be executed, this file is the standard ALGOL output file, channel 61.

Under INTERCOM, the files INPUT and OUTPUT are always disconnected during compilation. Under TELEX, these files are always connected.

The RUN command suppresses all compile-time listings to the OUTPUT file except fatal error listings and (unless SUP is specified with RUN) nonfatal error diagnostics. When fatal errors are encountered during assembly or compilation, the lines containing the errors are listed at the terminal together with the corresponding ALGEDIT line numbers and diagnostic messages. This information is not listed for nonfatal errors unless fatal errors have also been encountered. If no fatal errors have been encountered, execution begins unless NOEX has been specified with RUN.

When the edit file is to be run, it is copied automatically to a system file, which is then transferred to the compiler or assembler. The length of the lines copied to the system file is restricted by the format specification in effect at the terminal. All lines are blank-filled or truncated to conform to the maximum character count. If truncation occurs, a message indicating the length of the longest line encountered is displayed.

Examples:

To compile an ALGOL program contained in the edit file:

. . <u>RUN, A, N</u>	User enters RUN, ALGOL system-name, NOEX
JOB COMPILING	System issues a message.
. .	ALGEDIT is ready for the next command.

To compile and execute a FORTRAN Extended program contained in the local file XPROG, the executing program requests input data, processes data, and displays the results:

. . <u>RUN, F=XPROG, FTN</u>	User enters RUN, FILE=filename, system-name.
JOB COMPILING	System issues a message
ENTER 2-DIGIT NUMBER <u>13</u>	Program issues a request; user enters the data.
THE CUBE OF 13 IS 2197	Program displays the result of the computation.
. .	ALGEDIT is ready for the next command.

To assemble and execute a COMPASS program contained in the edit file (an arithmetic error terminates the job during execution):

. . <u>RUN, COMP</u>	User enters RUN, system-name.
JOB COMPILING	System issues a message.
ARITHMETIC ERROR MODE=1 ADDRESS=23427	System issues a message.
. .	ALGEDIT is ready for the next command.

17.9.2 INTERRUPTING EXECUTION

The user can suspend or terminate execution of a program by entering an interrupt command. An abort under INTERCOM returns the user to ALGEDIT command mode, and the contents of the edit file are retained. An abort under NOS 1 returns the user to NOS 1 interactive command mode, and the contents of the edit file are lost.

17.10 EXAMPLE

```

COMMAND- REDUCE,OFF ← Inhibit automatic field length reduction.
COMMAND- ALGEDIT.

ALGEDIT 01/10/75 14.44.52- VERSION: 1.0 L2
.. FORMAT,SHOW

WRNG- CH= 72 TAB CHAR=5 TAB COL= 7 10 13 16 19 . ← ALGEDIT displays tab stops.
.. CHECK

ERR- ON OR OFF REQUIRED.
.. CHECK,ON ← Turn on checking mode.

.. C

ERR- UNRECOGNIZABLE COMMAND.
.. CR ← Initiate text mode.

100 = "BEGIN"
110 = "INTEGER" N;
120 = N := 5;
130 = $"INTEGER" "ARRAY" FIBONACC(1:N, 1:2);
=> => =>LINE 130 THE DECLARATION "INTEGER"... SHOULD NOT OCC
UR AFTER STATEMENT(S)
140 = =125=$"BEGIN" ← Insert line.
140 = $"REAL" "ARRAY" PHI(2:N); ← ALGEDIT returns at current line.
=> => =>LINE 140 RECOVERY: ARRAY"PHI( REPLACED BY "ARRAY"PHI
[
150 = =140=$"REAL" "ARRAY" PHI(2:N); ← Correct typing error (ALGEDIT recovery
150 = $"INTEGER" K; leaves edit file unchanged).
160 = $FIBONACC(1,2) := 1 FIBONACC(1,1) := 1;
=> => =>LINE 160 MISSING DELIMITER BETWEEN <INTEGER NUMBER>
AND FIBONAC ... FIRST OPERAND DELETED
170 = =160=$FIBONACC(1,2) := 1; FIBONACC(1,1) := 1;
170 = $FIBONACC(2,2) := 1; FIBONACC(2,1) := 2;
180 = $PHI(2) := FIBONACC(2,2) / FIBONACC(1,2);
190 = $OUTPUT(61,("( /,5Z,10Z"),FIBONACC(1,1),FIBONACC(2,2));
200 = $OUTPUT(61,("( /,5Z,10Z,10Z.5D"),
210 = $FIBONACC(2,1),FIBONACC(2,2),PHI(2));
220 = $"FOR" K := 3 "STEP" 1 "UNTIL" N "DO"
=> => =>LINE 220 ILLEGAL ":" AFTER "FOR" IN FOR VARIABLE
230 = = ← Exit text mode.
.. /- / = / = / , 220 , V ← Correct by text replacement command.
220 = "FOR" K := 3 "STEP" 1 "UNTIL" N "DO" Y — ALGEDIT displays correction;
user verifies.
WRNG- 1: CHANGE(S)
.. CHECK,OFF ← Turn off checking mode.
.. ADD ← Reenter text mode.

```

```

230 =      $$"BEGIN"
240 =      $$FIBONAC[K,2] := FIBONAC[K-1,2] + FIBONAC[K-2,2];
250 =      $$FIBONAC[K,1] := K;
260 =      $$PHI[K] := FIBONAC[K,2] / FIBONAC[K-1,2];
270 =      $$OUTPUT(61, "(" / 5Z, 10Z, 10Z.5D")",
280 =      $$FIBONAC[K,1], FIBONAC[K,2], PHI[K]);
290 =      $$"END"
300 =      "END"
310 =      "EOP" ← Program is completed.
320 =      =

..      ANALYZE ← Analyze syntax of entire program.

=> => => LINE      310      ADVISORY:  END(S) MISSING FOR USER IMPLICIT
      OUTER BLOCK(S)... ADDED BY COMPILER
..      295=$$"END" ← Insert missing "END."
ANALYZE

      300 ="END"
=> => => LINE      300      ADVISORY:  YOUR PROGRAM ENDS HERE
..      RESEQ ← Resequence ALGEDIT line numbers in
                        edit file. List program.

..      LIST, ALL

100 ="BEGIN"
110 ="INTEGER" N;
120 =N := 5;
130 =      "BEGIN"
140 =      "INTEGER" "ARRAY" FIBONAC[1:N, 1:2];
150 =      "REAL" "ARRAY" PHI[2:N];
160 =      "INTEGER" K;
170 =      FIBONAC[1,2] := 1;      FIBONAC[1,1] := 1;
180 =      FIBONAC[2,2] := 1;      FIBONAC[2,1] := 2;
190 =      PHI[2] := FIBONAC[2,2] / FIBONAC[1,2];
200 =      OUTPUT(61, "(" / 5Z, 10Z")", FIBONAC[1,1], FIBONAC[2,2]);
210 =      OUTPUT(61, "(" / 5Z, 10Z, 10Z.5D")",
220 =      FIBONAC[2,1], FIBONAC[2,2], PHI[2]);
230 =      "FOR" K := 3 "STEP" 1 "UNTIL" N "DO"
240 =      "BEGIN"
250 =      FIBONAC[K,2] := FIBONAC[K-1,2] + FIBONAC[K-2,2];
260 =      FIBONAC[K,1] := K;
270 =      PHI[K] := FIBONAC[K,2] / FIBONAC[K-1,2];
280 =      OUTPUT(61, "(" / 5Z, 10Z, 10Z.5D")",
290 =      FIBONAC[K,1], FIBONAC[K,2], PHI[K]);
300 =      "END"
310 =      "END"
320 ="END"
330 ="EOP"

..      SAVE, FIB, 100, 320 ← Save program without "EOP."

```

.. RUN,ALGOL,F=FIB ← Compile and execute ALGOL program in local file.

COMPILATION-OPTIONS BEILNX
XXALGOL COMPILED
.642 CP SECONDS COMPILATION TIME

CHANNEL,60=INPUT,P80
CHANNEL,61=OUTPUT,P136,PP60

1	1	
2	1	1.00000
3	2	2.00000
4	3	1.50000
5	5	1.66667

END OF ALGOL 4.1 LEVEL 0002
.85 CP SECONDS EXECUTION TIME

.. /5=/15/,120,V ← Alter program for reexecution.

120 =N := 15;Y

WRNG- 1 CHANGE(S)

.. DELETE,200,220 ← Delete three consecutive lines.

.. 275=SS"IF" K "GREATER" N-5 "THEN"
ANALYZE ← Insert line in edit file; ALGEDIT responds with only a line feed.

320 ="END"

=> => =>LINE 320 ADVISORY: YOUR PROGRAM ENDS HERE

.. SAVE,FIB,100,320,0 ← Save program, overwriting previous contents of file.

.. RUN,ALGOL,F=FIB

COMPILATION-OPTIONS BEILNX
XXALGOL COMPILED
.474 CP SECONDS COMPILATION TIME

CHANNEL,60=INPUT,P80
CHANNEL,61=OUTPUT,P136,PP60

11	89	1.61818
12	144	1.61798
13	233	1.61806
14	377	1.61803
15	610	1.61804

END OF ALGOL 4.1 LEVEL 0002
.54 CP SECONDS EXECUTION TIME

.. BYE ← Exit ALGEDIT.

COMMAND-

STANDARD CHARACTER SETS

A

CONTROL DATA operating systems offer the following variations of a basic character set:

CDC 64-character set

CDC 63-character set

ASCII 64-character set

ASCII 63-character set

The set in use at a particular installation was specified when the operating system was installed.

Depending on another installation option, the system assumes an input deck has been punched either in 026 or in 029 mode (regardless of the character set in use). Under NOS/BE 1, the alternate mode can be specified by a 26 or 29 punched in columns 79 and 80 of the job statement or any 7/8/9 card. The specified mode remains in effect through the end of the job unless it is reset by specification of the alternate mode on a subsequent 7/8/9 card.

Under NOS 1, the alternate mode can be specified by a 26 or 29 punched in columns 79 and 80 of any 6/7/9 card, as described above for a 7/8/9 card. In addition, 026 mode can be specified by a card with 5/7/9 multipunched in column 1, and 029 mode can be specified by a card with 5/7/9 multipunched in column 1 and a 9 punched in column 2.

Graphic character representation appearing at a terminal or printer depends on the installation character set and the terminal type. Characters shown in the CDC Graphic column of the standard character set table are applicable to BCD terminals; ASCII graphic characters are applicable to ASCII-CRT and ASCII-TTY terminals.

STANDARD CHARACTER SETS

Display Code (octal)	CDC			ASCII		
	Graphic	Hollerith Punch (026)	External BCD Code	Graphic Subset	Punch (029)	Code (octal)
00 [†]	: (colon) ^{††}	8-2	00	: (colon) ^{††}	8-2	072
01	A	12-1	61	A	12-1	101
02	B	12-2	62	B	12-2	102
03	C	12-3	63	C	12-3	103
04	D	12-4	64	D	12-4	104
05	E	12-5	65	E	12-5	105
06	F	12-6	66	F	12-6	106
07	G	12-7	67	G	12-7	107
10	H	12-8	70	H	12-8	110
11	I	12-9	71	I	12-9	111
12	J	11-1	41	J	11-1	112
13	K	11-2	42	K	11-2	113
14	L	11-3	43	L	11-3	114
15	M	11-4	44	M	11-4	115
16	N	11-5	45	N	11-5	116
17	O	11-6	46	O	11-6	117
20	P	11-7	47	P	11-7	120
21	Q	11-8	50	Q	11-8	121
22	R	11-9	51	R	11-9	122
23	S	0-2	22	S	0-2	123
24	T	0-3	23	T	0-3	124
25	U	0-4	24	U	0-4	125
26	V	0-5	25	V	0-5	126
27	W	0-6	26	W	0-6	127
30	X	0-7	27	X	0-7	130
31	Y	0-8	30	Y	0-8	131
32	Z	0-9	31	Z	0-9	132
33	0	0	12	0	0	060
34	1	1	01	1	1	061
35	2	2	02	2	2	062
36	3	3	03	3	3	063
37	4	4	04	4	4	064
40	5	5	05	5	5	065
41	6	6	06	6	6	066
42	7	7	07	7	7	067
43	8	8	10	8	8	070
44	9	9	11	9	9	071
45	+	12	60	+	12-8-6	053
46	-	11	40	-	11	055
47	*	11-8-4	54	*	11-8-4	052
50	/	0-1	21	/	0-1	057
51	(0-8-4	34	(12-8-5	050
52)	12-8-4	74)	11-8-5	051
53	\$	11-8-3	53	\$	11-8-3	044
54	=	8-3	13	=	8-6	075
55	blank	no punch	20	blank	no punch	040
56	, (comma)	0-8-3	33	, (comma)	0-8-3	054
57	. (period)	12-8-3	73	. (period)	12-8-3	056
60	≡	0-8-6	36	#	8-3	043
61	[8-7	17	[12-8-2	133
62]	0-8-2	32]	11-8-2	135
63	% ^{††}	8-6	16	% ^{††}	0-8-4	045
64	⌘	8-4	14	" (quote)	8-7	042
65	↵	0-8-5	35	_ (underline)	0-8-5	137
66	↴	11-0 or 11-8-2 ^{†††}	52	! (exclamation)	12-8-7 or 11-0 ^{†††}	041
67	↵	0-8-7	37	&	12	046
70	↴	11-8-5	55	' (apostrophe)	8-5	047
71	↴	11-8-6	56	?	0-8-7	077
72	↴	12-0 or 12-8-2 ^{†††}	72	<	12-8-4 or 12-0 ^{†††}	074
73	↴	11-8-7	57	>	0-8-6	076
74	↴	8-5	15	@	8-4	100
75	↴	12-8-5	75	\	0-8-2	134
76	↴	12-8-6	76	˘ (circumflex)	11-8-7	136
77	; (semicolon)	12-8-7	77	; (semicolon)	11-8-6	073

[†]Twelve zero bits at the end of a 60-bit word in a zero byte record are an end of record mark rather than two colons.

^{††}In installations using a 63-graphic set, display code 00 has no associated graphic or card code; display code 63 is the colon (8-2 punch). The % graphic and related card codes do not exist and translations yield a blank (55p).

^{†††}The alternate Hollerith (026) and ASCII (029) punches are accepted for input only.

A number of COMPASS coded macros are provided to expand the areas of application of ALGOL programs. ALGOL programs can make use of subprograms coded in COMPASS (by using code) to perform tasks which require the compactness of extreme code efficiency.

These macros are subdivided into four groups: entry/exit macros, which provide the link between the ALGOL declaration of a procedure and the COMPASS subprogram; specification macros, which supply the means for analyzing the parameters passed to the COMPASS subprogram; formal handling macros, which enable certain operations with these parameters; and a macro for requesting local stack space.

The user subprogram can use any registers, since all registers used by the run time system are initialized within the macro expansions. Care must be taken, however, to save any registers that may be needed after the macro call.

The macros interfacing between ALGOL and COMPASS are part of the system text overlay ALGTEXT. To assemble a COMPASS subprogram to be called from an ALGOL program, therefore, S=ALGTEXT must be specified on the COMPASS control statement.

The following text uses the symbols { , , } to signify that there is a choice between the items enclosed within the braces and separated by commas.

B.1 ENTRY/EXIT MACROS

The following two macros provide entry from a procedure statement or function designator in an ALGOL main program, and return, with a value in the latter case, to the appropriate point in the calling program.

B.1.1 ALGOL

This macro provides a linkage between the code declaration of an external procedure in an ALGOL main program and the corresponding COMPASS subprogram. Control transfers to the ALGOL macro expansion, and the statements which immediately follow the macro constitute the first executable code in the procedure. The macro expansion creates a new program level in the stack and establishes the environment for the macros.

name ALGOL number

name Optional location field parameter, up to 7 characters. A name for the COMPASS subprogram initiated at this point. This name will be used during traceback in event of an execution time error. If omitted, it is assumed to be: C ↦ number.

number Optional, up to 5 digits. The code number used in the declaration of the associated procedure in the ALGOL main program. This number, given as C ↦ number, links the COMPASS subprogram with the ALGOL declaration.

If more than 7 characters are used for the name or more than 5 digits for the number, it is truncated and a warning message given. The name and the number are related as follows:

name	number	traceback name	link for loader
blank	blank	C00000	C00000
blank	number	C ↦ number	C ↦ number
name	blank	name	name
name	number	name	C ↦ number

Examples:

ALGOL declaration	Entry Macro
<u>procedure</u> A(x,y); <u>code</u> 0;	ALGOL
<u>procedure</u> A(x,y); <u>code</u> ;	A ALGOL
<u>procedure</u> A(x,y); <u>code</u> 100;	{ ALGOL 100 AAA ALGOL 100
<u>procedure</u> A(x,y); <u>code</u> PROCA;	PROCA ALGOL

B.1.2. RETURN

This macro is used for a normal return from a COMPASS coded procedure. Although the procedure may terminate in other ways (GOTO, ERROR, and XEQ macros, for instance) this is the normal point of exit, and returns control to the point in the main program from which the ALGOL macro was entered. If invocation was by a function designator, the value of the function must be present in an X register at this stage. RETURN also performs an optional conversion of the contents of the X register.

RETURN xreg,type

xreg Optional X register designator. This X register must contain the value of the function to be returned to the calling program. If xreg is omitted, it is assumed the code was executed from a procedure statement.

type Type of the value in xreg. The following types are valid:

- (R) Floating point to be normalized; R is the default type.
- (I) Floating point to become an integer[†] (in the ALGOL sense, i.e., entier (xreg+0.5))
- (B) Boolean to be passed straight through without any additional operation
- (F) Machine integer[†] to be converted to floating point and normalized

[†]Throughout this appendix, a distinction is made between the term machine integer (fixed point number) as used in COMPASS, and the ALGOL type integer, which is stored internally in floating point.

Examples:

RETURN		If called as procedure
RETURN	X4	If called as function. The result in X4 is assumed to be floating point and is to be normalized.
N EQU 5		
RETURN	X.N,B	If called as function. Result in X5 is Boolean (bit 59 determines value).

B.2. SPECIFICATION MACROS

The following macros are used within a COMPASS coded procedure to establish the number, kinds, and types of actual parameters in the current invocation of the procedure.

B.2.1. PARAMS

This macro specifies the number of actual parameters in the current invocation of the COMPASS procedure. No check is performed on the parameter count.

PARAMS	xreg	
xreg		X register designator. The X register receives the number (in machine integer form) of actual parameters transmitted in the current call.

Example:

PARAMS	X5
--------	----

B.2.2. KIND

This macro provides a numerical representation of the kind of an actual parameter. No check is made on the parameter.

KIND	param,xreg	
param		{Absolute address expression, X register designator}. The value or contents of param is the ordinal of the actual parameter to be examined; 2 indicates the second parameter, for example.

xreg X register designator. The X register receives the numerical kind code (in machine integer form) of the specified actual parameter. The register can be the same as that specified by param. The codes corresponding to each kind of actual parameter are as follows:

switch	0	simple variable	8
string	1	subscripted variable	9
label	2	virtual array	10
no type procedure	3	no parameter, no type procedure	11
typed procedure	4	no parameter, typed procedure	12
array	5		
constant	6		
expression	7		

Examples:

```

FORMAT    EQU        5
          SX2        4
          KIND       FORMAT,X6
          KIND       3,X1
          KIND       X2,X3
          KIND       X2,X2

```

B.2.3 TYPE

This macro provides a numerical representation of the type of an actual parameter. No check is made on the parameter.

TYPE param,xreg

param { Absolute address expression, X register designator } . The value or contents of param is the ordinal of the actual parameter to be examined.

xreg X register designator. The X register receives the numerical type code (in machine integer form) of the specified actual parameter and can be the same register as that specified by param. The type codes are as follows:

no type	0
integer	1
real	2
Boolean	3

Examples:

```

TYPE       2,X5
TYPE       X3,X3
TYPE       ACTUAL,X7

```

B.2.4. SPEC

This macro permits the programmer to restrict the kinds of actual parameters. Since allowable kinds for a given parameter may be run-time dependent, this check occurs in line. This macro may produce abnormal termination with a diagnostic, and it does not provide direct information about the kind of the actual parameter. Any number of parameters may be checked with a single call to this macro; the only limit is imposed by COMPASS, which allows up to 9 continuation lines per statement.

SPEC (par₁:spc₁, par₂:spc₂, . . . , par_n:spc_n)

par₁

par₂ . . . par_n

Absolute address expression. The value or contents of par_i is the ordinal of the actual parameter to be examined. Only par₁ may be an X register; the second and subsequent par_i must be absolute address expressions.

:

Separates par_i part from the spc_i part.

spc_i

String of 1 to 12 single character spec codes, where each is an alphabetic character representing an allowable kind. Only spec codes may appear in the string, and none more than once. The string defines all allowable kinds for this parameter. The spec codes for each kind of actual parameter are:

switch	W
string	G
label	L
no type procedure	P
typed procedure	F
array	A
constant	C
expression	X
simple variable	V
subscripted variable	S
virtual array	R
no parameter, no type procedure	N
no parameter, typed procedure	T

Since there are 13 kinds, the string may have up to 12, to avoid checking when all kinds are allowed.

If the actual parameter is not one of the specified kinds, the diagnostic PARAMETER KIND ERROR is produced and the program aborts (see B.3).

Examples:

SX1	2	The second actual parameter must be a constant, expression, simple variable, subscripted variable or typed procedure without parameter; the fifth parameter must be a simple variable or subscripted variable.
SPEC	(X1: CXVST, 5: VS)	
SPEC	(3: AV, X2: W)	This macro call is erroneous because the second pair contains an X register.

B.3. FORMAL HANDLING MACROS

Any kind of actual parameter may be passed to the COMPASS subprogram, and subsequent handling of them as formal parameters is the responsibility of the subprogram. None of the macros in this section is applicable to all kinds of parameters. The most efficient way to ensure correct execution is by use of the macros KIND and SPEC. An alternative method is provided by optional checking within the macro itself to ensure that the parameter kind is consistent with the specified operation. This automatic checking is less efficient since it must be processed inline to maintain flexibility, and the same parameter might be checked several times if it is used by different macros.

B.3.1. VALUE

This macro causes the current value of a parameter to be computed. It is applicable to the following kinds of parameters only: constant (6), expression (7), simple variable (8), subscripted variable (9) or no-parameters-typed procedure (12) (in other words, CXVST). The parameter is optionally checked for kind.

VALUE	param, xreg, kind, check
param	{ Absolute address expression, X register designator } . Indicates the ordinal of the parameter.
xreg	X register designator. The X register receives the value of the indicated actual parameter. The value may be a floating point quantity in the case of real or integer. If Boolean only bit 59 is used (see 5.1.3).
kind	Optional, absolute address expression. Numerical kind code of the actual parameter, if known. This specification produces a more efficient expansion. It is ignored if its value cannot be ascertained at assembly time or if the check parameter is present.
check	Optional. If this specification is present (nonblank) the actual parameter is tested for kinds of parameters allowed in this macro. Abnormal termination with a PARAMETER KIND ERROR diagnostic may occur at execution time. If the kind parameter is nonnull, kind is ignored, an informative message is issued at assembly time, and the macro is expanded as for the most general case.

Examples:

VALUE 3,X1,,CHECK	Evaluates the third parameter into X1. Checks that the kind is correct.
VALUE 5,X2,6	Evaluates the fifth parameter into X2, assuming that it is a constant (6).
SX3 1 VALUE X3,X3	Evaluates the first parameter into X3.
B EQU 6 VALUE B,X6,8,C	Evaluates the sixth parameter into X6, assuming that it is a simple variable (8).

B.3.2. ASSIGN

This macro is applicable only to a parameter which is either a simple variable (8) or a subscripted variable (9) (i.e., VS); and it optionally checks for these kinds. It assigns the contents of an X register to the specified parameter. Care must be taken to provide either a normalized floating point quantity or a Boolean value in the X register. Use of the macro XFORM (see B.3.7) is advised for this purpose. Assignments take place only into stack addresses in SCM/CM; assigning into a subscripted variable in LCM/ECS not only does not produce the correct result, but destroys the contents of an SCM/CM address.

ASSIGN	param,xreg,kind,check
param	{ Absolute address expression, X register designator }. See B.3.1.
xreg	X-register designator. X register contains the value to be assigned to the specified actual parameter.
kind	Optional, absolute address expression. See B.3.1.
check	Optional. See B.3.1

Examples:

SX1	3	Assigns the value 3 (after converting it to floating point) to the fifth parameter. Checks correctness of kind.
XFORM	X1,X1,FR	
ASSIGN	5,X1,8,C	
N	MX2 1	Assigns the Boolean value <u>true</u> to the second parameter after checking that it is the correct kind.
SET	2	
ASSIGN	N,X2,,C	

B.3.3. ADDRESS

This macro is applicable to the following kinds of parameters only: a string (1), an array (5), a constant (6), a simple variable (8), a subscripted variable (9) or a virtual array (10), (that is, GACVSR). Optionally, it checks for these kinds. It calculates the address, or first word address, of the specified parameter. The macro also tells whether the computed address is in CM/SCM or in ECS/LCM.

Arrays and strings always grow from RA toward RA+FL, as opposed to the stack which grows from RA+FL toward RA.

ADDRESS	param,xreg,kind,check
param	{ Absolute address expression, X register designator } . See B.3.1.
xreg	X register designator. The X register receives in the lower 18 bits the address [†] of a constant or a variable, or the first word address of an array or a string. Bit 59 will be 1 if the address is in LCM/ECS and 0 if in SCM/CM (for arrays only).
kind	Optional. Absolute address expression. See B.3.1.
check	Optional. See B.3.1.

Examples:

ADDRESS	3,X1,5	Returns in X1 the first word address of the third parameter, assumed to be an array (5).
SX2	4	Returns in X3 the address of the fourth parameter and checks correctness of kind.
ADDRESS	X2,X3,,C	

B.3.4. LENGTH

This macro is applicable to the following kinds of parameters: string (1), array (5), or virtual array (10). Optionally, it checks kind. It calculates the number of elements in the array or the number of characters in a string.

LENGTH	param,xreg,kind,check
param	{ Absolute address expression, X register designator } . See B.3.1.
xreg	X register designator. X register receives the length of the actual parameter in machine integer form.
kind	Optional. Absolute address expression. See B.3.1.
check	Optional. See B.3.1.

[†]Only 18-bit addresses are allowed for LCM.

A string is stored in the coefficient part of a word with zero exponent, eight characters per word, and the count includes at least six characters for the initial and terminal string quotes.

Examples:

Y	EQU LENGTH	3 Y,X2,1	Return in X2 the number of characters in the string (1) given by the third parameter.
	LENGTH	4,X4,10	Returns in X4 the length of the virtual array (10) given by the fourth parameter.
	SX1 LENGTH	1 X1,X7,,C	Returns in X7 the length of the array or string given by the first parameter after checking correctness of kind.

B.3.5. ORDER

This macro operates only on array (5) or virtual array (10) parameters. It returns the order (dimensionality) of the array (the number of subscripts required to address the array in its original declaration). Optionally, it checks that the kind is correct.

ORDER	param,xreg,check
param	{ Absolute address expression, X register designator }. See B.3.1.
xreg	X register designator. The X register receives, in machine integer form, the order of the specified actual array parameter.
check	Optional. See B.3.1.

Examples:

```
ORDER    A, X2, CHECK
ORDER    X3,X7
```

B.3.6. GOTO

This macro applies only to a switch or a label parameter (or a designational expression), and optionally checks kind. It causes a transfer to a label outside the COMPASS subprogram in accordance with the rules applying to the go to statement, and it can be used on the occurrence of abnormal conditions within the COMPASS procedure. If the actual parameter is a switch (not a switch element), the value of the index for the required switch element must be supplied in the macro call.

GOTO param, index, check

param {Absolute address expression, X register designator} . See B.3.1.

index Optional. {Absolute address expression, X register designator} . If omitted, the actual parameter is assumed to be a label, and control is transferred to that point. If present, the actual parameter is assumed to be a switch, and the value or contents of index (machine integer form) must be the subscript of the required switch. The switch element is found after checking that the index is within the bounds of the switch (otherwise a fatal error occurs) and control is transferred accordingly.

check Optional. See B.3.1. The specified actual parameter is checked to ascertain if it is a label or a switch depending on the absence or presence of index.

Examples:

GOTO 3, ,CCC Checks whether the third parameter is a label and transfers control to it.

SX1 5 Transfers control to the second switch element of the switch given by the fifth
GOTO X1,2 parameter.

B.3.7. XFORM

This macro provides for the conversion of quantities into various data formats.

XFORM operand, result, action

operand X register designator. The X register contains the quantity to be transformed according to the action.

result X register designator. The X register receives the result of the conversion. The same register may be used as operand.

action { RI, RF, FR } .
If RI, the operand is assumed to be a real and the result will be an integer in the ALGOL sense, that is:

result:=entier (operand+0.5)

If RF, the operand is a floating point quantity (real or integer) and the result will be a machine integer.

If FR, the operand is a machine integer quantity and the result will be a floating point number.

Examples:

SX2	3	The contents of X3 will be the value 3 expressed as a normalized
XFORM	X2,X3,FR	floating point quantity.
XFORM	X6,X6,RI	The contents of X6 before the macro call is assumed to be a floating point
		real quantity. After the macro call it will be a floating point ALGOL integer
		quantity.

B.3.8. STRING

This macro transforms a given COMPASS string into a string suitable for ALGOL and builds its descriptor. The resulting string may be passed as an actual parameter to an ALGOL procedure via the macro XEQ (See B.3.10).

STRING	name, string
name	A COMPASS symbol. This name is used by the macro as the symbol to define the address where the descriptor is placed, hence it must not be used anywhere in the location field in the COMPASS subprogram.
string	The string may contain any character from the character set, except \$. It may be formed by any number of characters, including zero (the empty string), and is delimited by parentheses.

Usage of this macro will produce the nonfatal COMPASS error number 9 because it is not possible to change the micro marks (\neq) of COMPASS, and they are used in ALGOL for opening and closing a string:
 $\neq(\neq$ THIS IS NOT A MICRO $\neq)\neq$

Examples:

STRING	MYSTRG,(THIS STRING CONTAINS . , : ; [])
STRING	A, (/3B,-Z.DDDBDDD)
STRING	B This string is empty

B.3.9. ERROR

This macro prints a given string and passes control to the routines which produce the traceback, according to the abnormal termination dump format option. (See 8.2.) Then the program aborts.

ERROR	string
string	The string may contain any character from the character set, except \$. The number of characters is limited to 135 and the empty string is not allowed. The string is delimited by parentheses.

Example:

ERROR	(THIS IS AN ERROR MESSAGE)
-------	----------------------------

B.3.10. XEQ

This macro operates only on an actual parameter that is a no-type procedure with parameters (3) or without (13). The macro invokes this formal procedure from within the COMPASS subprogram. It permits the COMPASS subprogram to pass parameters to the procedure but they may be only simple variables (of types real, integer, or Boolean), strings defined by the macro STRING, or alternatively other formal parameters of the COMPASS subprogram. With the first type of parameter (3), the subprogram can transmit simple values to the formal procedure and receive results after execution. These parameters are local to the COMPASS subprogram. With the last type of parameter (13), more complex kinds (arrays, switches, labels, procedures) can be transmitted from the original main program to the formal procedure via the COMPASS subprogram.

The XEQ macro always produces code to check that the specified actual parameter is a no-type procedure.

According to ALGOL up to 63 parameters can be passed to the formal procedure. The only limit imposed by the macro is that imposed by COMPASS, which allows up to 9 continuation lines per statement (see 5.4.3).

XEQ proc, (P1,P2, . . . , Pn)

proc { Absolute address expression, X register designator } . The value, or contents, of proc is the ordinal of the no type procedure parameter. This parameter is referred to as the formal procedure.

P1, . . . , Pn Optional. If omitted, the formal procedure is assumed to be a no-parameter procedure, and control will be given to it immediately.

If given, Pi may have two forms:

param : typknd

param { Relocatable address expression, name given to a string } . If it is a relocatable address expression, param is assumed to be the address of a variable local to the COMPASS subprogram which provides an input value to the formal procedure or expects to receive a result from it. If it is a name for a string use must have been made of the macro STRING.

: Separator.

typknd { R,I,B,S } . The variable local to the COMPASS subprogram is of type real (R), integer (I) or Boolean (B). For a string, S must be used.

*param

* The asterisk means a special parameter.

param { Absolute address expression, X register designator } . The value or contents of param is the ordinal of the original parameter to the COMPASS subprogram which is to be transmitted to the formal procedure.

Examples:

```

W   DATA   3.5
Y   VFD     1/1,59/0
Z   BSSZ    1

XEQ   3, (W:R,Y:B,*5,Z:R)

```

Checks that the third parameter to the COMPASS subprogram is a no-type procedure. It transmits the real variable W, the Boolean variable Y, the 5th parameter (to the COMPASS subprogram) and the real Z (where the procedure in question expects a result) as the 1st, 2nd, 3rd, and 4th parameters respectively to the formal procedure given. Then the procedure is executed.

```

Q   EQU     10
    SX1     4
    SX2     6
    STRING  A,(/6ZDD.DDD)
    XEQ     X1,(*Q,*X2,A:S)

```

Checks that the 4th parameter to the COMPASS subprogram is a no-type procedure. It transmits its 10th parameter, 6th parameter and the string named A as the 1st, 2nd, and 3rd parameters, respectively, to the formal procedure and executes it.

```

XEQ   5

```

Checks that the 5th parameter to the COMPASS subprogram is a no-type procedure; and since there are no parameters, it executes it as a procedure without parameters.

B.4. STACK REQUEST MACRO (GETSCM)

This macro grants a given number of words in the local stack assigned to the COMPASS subprogram. The user requests this amount of core and provides a register where the first word address for the stack space is set.

The stack grows from RA+FL towards RA; therefore, this area must be used by indexing the first word address negatively.

This macro must be used with extreme care, otherwise the stack and the code are liable to undetected destruction. Additionally, the space granted may be preset in accordance with the execution-time P option. (See 8.3).

```

GETSCM    size,fwa

```

* size { X register designator, absolute address expression } . The contents or value of size indicate the number of words to be requested.

fwa X register designator. This X register will have the first word address of the granted stack space. It can be the same register as size. This area comprises from fwa to fwa-size+1 (from RA+FL towards RA).

Examples:

SX5	7	
GETSCM	X5,X6	In both cases 7 words of stack are requested; and when granted, the first word
GETSCM	7,X6	address will be found in X6.

B.5. EXAMPLES

In the following examples, both the calling ALGOL program and the COMPASS subprogram are given.

B.5.1 EXAMPLE 1

The calling ALGOL program is:

```
'BEGIN' 'COMMENT' THIS PROGRAM WILL PRINT THE VALUES 1 2 3 4 IF THE COMPASS
PROCEDURE JUMP IS CORRECT, OTHERWISE VALUES LIKE 300, 400, 500 AND
1000 WILL APPEAR;
```

```
'SWITCH' S:=L5,L4,L3;
'PROCEDURE' JUMP(A,B,C);
    'LABEL' A; 'SWITCH' B; 'BOOLEAN' C; 'CODE' 100;
        OUTREAL(61,1);
        JUMP(L1,S, 'TRUE');
        OUTREAL(61,300);
L1:    OUTREAL(61,2);
        JUMP(L2,S,'FALSE');
L5:    OUTREAL(61,500);
        'GOTO' END;
L4:    OUTREAL(61,400);
        'GOTO' END;
L2:    OUTREAL(61,1000);
        'GOTO' END;
L3:    OUTREAL(61,3);
END:   OUTREAL(61,4)
'END'
```

The COMPASS subprogram is:

```
IDENT JUMPS
SST
```

```
*THIS PROGRAM TESTS THE 3RD PARAMETER (A BOOLEAN). IF IT IS 'TRUE' WILL GIVE
*CONTROL TO THE LABEL (1ST PARAMETER). IF IT IS 'FALSE' WILL GIVE CONTROL TO
*THE THIRD ELEMENT IN THE SWITCH LIST SPECIFIED BY THE 2ND PARAMETER
```

	ALGOL	100	
	SPEC	(1:L,2:W,3:CV)	
	VALUE	3,X7	
	PL	X7,FALSE	
	GOTO	1	.A LABEL
	JP	FIN	
FALSE	GOTO	2,3	.A SWITCH
FIN	RETURN		
	END		

B.5.2. EXAMPLE 2

The calling ALGOL program is:

```
'BEGIN' 'COMMENT' THIS PROGRAM WILL OUTPUT THE 10 ELEMENTS OF THE ARRAY R, WHERE
THE COMPASS PROCEDURE TEST HAS STORED THE FOLLOWING ITEMS:
NUMBER OF PARAMETERS IN THE CALL (8), KIND OF A (0), KIND OF D (3),
KIND OF G (6), TYPE OF B (0), TYPE OF E (2), TYPE OF H (2), LENGTH OF B
(10), LENGTH OF F (10), ORDER OF F (1).
FINALLY Q IS ALSO OUTPUT (100);

'REAL' P;
'PROCEDURE' ONE(A); 'REAL' A; A:=A;
'REAL' 'PROCEDURE' TWO(A); 'REAL' A; TWO:=A;
'PROCEDURE' TEST (A,B,C,D,E,F,G,H);
    'SWITCH' A; 'STRING' B; 'LABEL' C;
    'PROCEDURE' D; 'REAL' 'PROCEDURE' E; 'ARRAY' F; 'REAL' G,H;
    'CODE';
'SWITCH' S:=L1,L2;
'ARRAY' R[1:10];
'REAL' Q;
L1:TEST(S,('TEST'),L2,ONE,TWO,R,100,Q);
L2:OUTARRAY(61,R);
    OUTREAL(61,Q);
'END'
```

The COMPASS subprogram is:

IDENT	PASSPAR
SST	

*THE FOLLOWING MACRO STORES A REAL QUANTITY IN SUCCESSIVE ARRAY ELEMENTS

STORE	MACRO		
*	XFORM	X7,X6,FR	PUT IN X6 THE FLOATING POINT VALUE OF THE FIXED POINT QUANTITY ORIGINALLY IN X7.
	SA1	FWAARR	
	SA6	X1	
	SX7	X1+1	
	SA7	FWAARR	.STORE INTO ARRAY
	ENDM		

TEST	ALGOL	
	SPEC	(1:W,2:G,3:L,4:PFNT,5:FT,6:A,7:CXVST,8:VS)
	ADDRESS	6,X6,5
	SA6	FWAARR .STORE FWA FOR ARRAY
	PARAMS	X7
	STORE	
	KIND	1,X7
	STORE	
	KIND	4,X7
	STORE	
	KIND	7,X7
	STORE	
	TYPE	2,X7
	STORE	
	TYPE	5,X7
	STORE	
	TYPE	8,X7
	STORE	
	VALUE	7,X6,6
	ASSIGN	8,X6
	LENGTH	2,X7,1
	STORE	
	LENGTH	6,X7,5
	STORE	
	ORDER	6,X7
	STORE	
	RETURN	
FWAARR	DATA	0
	END	

HARDWARE REPRESENTATION OF ALGOL SYMBOLS C

Table C-1 lists the ALGOL basic symbols.

The first column gives the symbols in the reference language (see section 2.1).

The second column gives the symbols in the standard representation of the hardware language. The characters listed are from the CDC character set; note that certain symbols are printed as different characters at installations using an ASCII character set. Both these character sets, together with the internal hardware representation of the character sets in CDC display code, are listed in Appendix A.

The third column gives (except for basic symbols underlined in the reference language) the keypunch (026 mode) that is converted to the standard hardware representation.[†] Refer to Appendix A for conversions from an 029 keypunch.

The fourth column gives, for certain basic symbols, an alternate hardware representation. These representations are accepted by ALGOL 4 to provide compatibility for programs written for an earlier CDC ALGOL compiler which was limited to a 48-character set.

[†]On some keypunches, the ' (apostrophe) key results in the ≠ delimiter.

TABLE C-1. HARDWARE REPRESENTATION OF ALGOL SYMBOLS

ALGOL Symbol	Standard Representation	Keypunch [†] (026)	Alternate Representation
A thru Z	A thru Z	12-1 thru 0-9	
a thru z	A thru Z	12-1 thru 0-9	
0 thru 9	0 thru 9	0 thru 9	
+	+	12	
-	-	11	
×	*	11-8-4	
/	/	0-1	
÷	//	0-1, 0-1	≠/≠ or ≠DIV≠
↑	↑ or ** (or * ^{††})	11-8-5 or 11-8-4, 11-8-4 (or 11-8-4 ^{††})	≠POWER≠
<	<	12-0	≠LESS≠
≤	≤	8-5	≠NOT GREATER≠
=	=	8-3	≠EQUAL≠
≠	¬ =	12-8-6, 8-3	≠NOT EQUAL≠
≥	≥	12-8-5	≠NOT LESS≠
>	>	11-8-7	≠GREATER≠
^	^	0-8-7	≠AND≠
∨	∨	11-0	≠OR≠
≡	≡	0-8-6	≠EQUIV≠
¬	¬	12-8-6	≠NOT≠
⊃	≠IMPL≠		

[†]On some keypunches, the key resulting in the ≠ delimiter (8-4 punch) is labeled with a '1'.

^{††}Only when used in a format string.

TABLE C-1. HARDWARE REPRESENTATION OF ALGOL SYMBOLS (Continued)

ALGOL Symbol	Standard Representation	Keypunch [†] (026)	Alternate Representation
.	.	12-8-3	
,	,	0-8-3	
:	:	8-2	.. or % ^{††††}
;	;	12-8-7	.,
10	≠	8-4	→
	blank	space	
((0-8-4	
))	12-8-4	
[[8-7	(/
]]	0-8-2	/)
:=	:=	8-2, 8-3	. = or .. =
' ^{†††}	≠(≠	8-4, 0-8-4, 8-4	
', ^{†††}	≠)≠	8-4, 12-8-4, 8-4	
<u>algol</u> ^{††}	≠ALGOL≠		
<u>array</u>	≠ARRAY≠		
<u>begin</u>	≠BEGIN≠		
<u>Boolean</u>	≠BOOLEAN≠		
<u>code</u> ^{††}	≠CODE≠		
<u>comment</u>	≠COMMENT≠		
<u>do</u>	≠DO≠		

[†] On some keypunches, the key resulting in the ≠ delimiter (8-4 punch) is labeled with a ' .

^{††} Not defined in the ALGOL-60 Revised Report; code and algol are defined in section 5.4.1., Chapter 2; eop in Chapter 4.

^{†††} The symbols ' and ' stand for open and close string quote symbols.

^{††††} The % is only valid for 64 character set.

TABLE C-1. HARDWARE REPRESENTATION OF ALGOL SYMBOLS (Continued)

ALGOL Symbol	Standard Representation	Keypunch [†] (026)	Alternate Representation
<u>else</u>	≠ELSE≠		
<u>end</u>	≠END≠		
<u>eop</u> ^{††}	≠EOP≠		
<u>false</u>	≠FALSE≠		
<u>for</u>	≠FOR≠		
<u>fortran</u> ^{††}	≠FORTRAN≠		
<u>go to</u>	≠GO TO≠		
<u>if</u>	≠IF≠		
<u>integer</u>	≠INTEGER≠		
<u>label</u>	≠LABEL≠		
<u>procedure</u>	≠PROCEDURE≠		
<u>own</u>	≠OWN≠		
<u>real</u>	≠REAL≠		
<u>step</u>	≠STEP≠		
<u>string</u>	≠STRING≠		
<u>switch</u>	≠SWITCH≠		
<u>then</u>	≠THEN≠		
<u>true</u>	≠TRUE≠		
<u>until</u>	≠UNTIL≠		
<u>value</u>	≠VALUE≠		
<u>while</u>	≠WHILE≠		
[†] On some keypunches, the key resulting in the ≠ delimiter (8-4 punch) is labeled with a ' . ^{††} Not defined in the ALGOL-60 Revised Report; code, algol, and fortran are defined in section 5.4.1., Chapter 2, eop in Chapter 4.			

The following two programs are identical. The first is coded in the standard hardware representation, and the second is coded in the alternate hardware representation.

TWO-DIMENSIONAL ARRAY: "BEGIN" "INTEGER" I;
 "COMMENT" THIS PROGRAM DECLARES A SERIES OF ARRAYS OF EVER-
 INCREASING DIMENSION. THE ARRAY IS THEN FILLED WITH COMPUTED
 VALUES, ONE OF WHICH IS ALTERED. THE ALTERED VALUE IS THEN
 SEARCHED FOR AND PRINTED.
 THE PROGRAM HALTS WHEN THE DECLARED ARRAY SIZE EXCEEDS THE
 AVAILABLE MEMORY. WHEN THIS OCCURS, THE PROGRAM EXITS WITH
 THE MESSAGE STACK OVERFLOW ON THE STANDARD
 OUTPUT UNIT;

```

      I:=10;
L:    I:=I+1;
      OUTPUT(61,("(/,3D)",I);
      "BEGIN" "ARRAY" A(-3*I:-1, I:2*I); "INTEGER" P,Q;
      "FOR" P:=-3*I "STEP" 1 "UNTIL" -1 "DO"
      "FOR" Q:=1 "STEP" 1 "UNTIL" 2*I "DO"
          A(P,Q):=-P+100*Q;
          A(-2*I,I+2):=A(-2*I,I+2) + 10000;
      "FOR" P:=-3*I "STEP" 1 "UNTIL" -1 "DO"
      "FOR" Q:=1 "STEP" 1 "UNTIL" 2*I "DO"
          "IF" A(P,Q)≠ 100*Q-P "THEN"
              "BEGIN" OUTPUT (61,("(/,5D)",A(P,Q)) "END";
              "GOTO" L
          "END"
      "END"
"END"

```

TWO-DIMENSIONAL ARRAY.. "BEGIN" "INTEGER" I.,
 "COMMENT" THIS PROGRAM DECLARES A SERIES OF ARRAYS OF EVER-
 INCREASING DIMENSION. THE ARRAY IS THEN FILLED WITH COMPUTED
 VALUES, ONE OF WHICH IS ALTERED. THE ALTERED VALUE IS THEN
 SEARCHED FOR AND PRINTED.
 THE PROGRAM HALTS WHEN THE DECLARED ARRAY SIZE EXCEEDS THE
 AVAILABLE MEMORY. WHEN THIS OCCURS, THE PROGRAM EXITS WITH
 THE MESSAGE STACK OVERFLOW ON THE STANDARD
 OUTPUT UNIT.,

```

      I..=10.,
L..   I..=I+1.,
      OUTPUT(61,("(/,3D)",I)..,
      "BEGIN" "ARRAY" A(/-3*I..-1, I..2*I/).., "INTEGER" P,Q.,
      "FOR" P..=-3*I "STEP" 1 "UNTIL" -1 "DO"
      "FOR" Q..=1 "STEP" 1 "UNTIL" 2*I "DO"
          A(/P,Q/)..=-P+100*Q.,
          A(/-2*I,I+2/)..=A(/-2*I,I+2/) + 10000.,
      "FOR" P..=-3*I "STEP" 1 "UNTIL" -1 "DO"
      "FOR" Q..=1 "STEP" 1 "UNTIL" 2*I "DO"
          "IF" A(/P,Q/) "NOT EQUAL" 100*Q-P "THEN"
              "BEGIN" OUTPUT (61,("(/,5D)",A(/P,Q/)) "END"..,
              "GOTO" L
          "END"
      "END"
"END"

```


STANDARD PROCEDURES FOR VECTOR AND MATRIX MANIPULATIONS

D

A set of built-in procedures is provided for certain operations on arrays. These procedures take full advantage of the characteristics of the instruction stack. Each procedure is essentially equivalent to ALGOL for-statements as shown in the definitions of these procedures.

The provision of the array procedures invests the computer with pipelined functional streaming units at the level of the source language. By defining the machine code for the macro expansions so that it loops wholly within the instruction stack, an emulation of pipelining is achieved with very fast execution speed. Only those array procedures whose machine code can fit within the instruction stack are provided. There is no general user macro definition facility at the source language level at present.

The use of these procedures produces execution speed gains on all machine models due to the compacting of the code. However, such gains are more significant for machines with an instruction stack, where they can be very appreciable. In all cases, a vector must be of length 10 or greater before gains are appreciable (see chapter 12, section 12.3).

D.1 MATRIX AND VECTOR PROCEDURES

The designs of the matrix procedures are given here in terms of ALGOL procedures, which does not imply that they are so implemented.

In the examples, upb and lwb are used to represent the upper and lower bounds of an array. They do not represent ALGOL procedures and are only used to facilitate the explanation.

D.1.1 MATRIX PROCEDURE

MATMULT: The standard procedure MATMULT calculates the crossed product of two matrices M1 (m,n) by M2 (n,p) giving the matrix M3 (m,p). The bound pairs are assumed to be compatible, or else a run-time error occurs. Arrays M1, M2 and M3 must all have 2 dimensions.

```
procedure MATMULT (M1,M2,M3,M,N,P); value M,N,P; integer M,N,P; array M1,M2,M3;  
  begin integer I,J,K,L1,L2,L3,R1,R2,R3;  
    real S;
```

```
  if M < 1 ∨ N < 1 ∨ P < 1 then ERROR ('MATMULT – NEGATIVE DIMENSIONS');
```

```
    L1 := lwb (M1,1);      R1 := lwb (M1,2);  
    L2 := lwb (M2,1);      R2 := lwb (M2,2);  
    L3 := lwb (M3,1);      R3 := lwb (M3,2);
```

```

if  ((upb (M3,1) - L3+1)  $\neq$  M)
   $\vee$  ((upb (M1,2) - R1+1)  $\neq$  N)
   $\vee$  ((upb (M2,2) - R2+1)  $\neq$  P) then ERROR ('MATMULT - INCONSISTANT MATRIX SIZES');

for  I:=0 step 1 until M-1 do
  for  J:=0 step 1 until P-1 do
    begin S:=0;
      for  K:=0 step 1 until N-1 do
        S:=S + M1 [L1+I, R1+K] *M2 [L2+K, R2+J];
        M3 [L2+I, R2+J]:=S
      end
    end
end MATMULT

```

D.1.2 VECTOR PROCEDURES

By vector is meant a one-dimensional array, the type of which is either Boolean or real (types must be compatible within each procedure).

The designs of the procedures are given, in terms of ALGOL procedures, below. The algorithms in the special case where lower-bound = 0 and upper-bound = N for each vector, are as follows:

VINIT (OP,E,S,V1)

build constant vector: $OP \leq 1$; $V1[I] := E$, I from 0 to N
arithmetic progression: $OP = 2$; $V1[0] := E$; $V1[I] := V1[I-1] + S$, I from 1 to N
geometric progression: $OP \geq 3$; $V1[0] := E$; $V1[I] := V1[I-1] * S$, I from 1 to N

VXMIT (OP, V1, V2, INIT, STEP);

$OP \leq 1$ transfers V1 to part of V2: $V2[INIT+i*STEP] := V1[i]$

$OP \geq 2$ transfers part of V2 to V1: $V1[i] := V2[INIT+i*STEP]$

VXMIT is a function that transfers a row or a column of a matrix to or from a vector.

VMONOD (OP,V1, V2)

$V2[i] := OP\ V1[i]$ where OP is a monadic operation as follows:

$OP \leq 1$	transmit	
$= 2$	reversed transmit	
$= 3$	real to integer transform	
$= 4$	negate	
$= 5$	absolute	
$= 6$	entier (floor)	
$= 7$	ceiling	
$= 8$	sign	
$= 9$	delta	$V2[i] := V1[i+1] - V1[i]$
≥ 10	mean	$V2[i] := (V1[i+1] + V1[i])/2$

VDIAD (OP, V1,V2,V3)

$V3[i] := V1[i] \text{ OP } V2[i]$ where OP is a diadic operation as follows:

OP ≤ 1 add
 = 2 subtract
 = 3 multiply
 = 4 divide
 = 5 average $V3[i] := (V1[i] + V2[i])/2$
 ≥ 6 average difference $V3[i] := (V1[i] - V2[i])/2$

VSDIAD (OP, E, V2, V3)

$V3[i] := E \text{ OP } V2[i]$ where E is a real expression and OP as for VDIAD. E is evaluated only once, on entry to the procedure.

VPROSUM (OP, V1)

sigma: VPROSUM:= $V1[0] + V1[1] + \dots + V1[N]$ when $OP \leq 1$
 product: $V1[0] * V1[1] * \dots * V1[N]$ when $OP \geq 2$

VDOT (V1,V2)

dot product: $VDOT := V1[0] * V2[0] + V1[1] * V2[1] + \dots + V1[N] * V2[N]$

BNOT (S1,S2)

monadic operation: $S2[i] := \neg S1[i]$

VBOOL (OP, S1,S2,S3)

$S3[i] := S1[i] \text{ OP } S2[i]$ where OP is a diadic operation as follows:

OP ≤ 1 or OP = 7 pierce
 = 2 and ≥ 8 inhibit
 = 3 imply
 = 4 equivalence
 = 5 exclusive or
 = 6 stroke

The truth table for Boolean operations is (0 stands for false and 1 for true):

Source		\vee	\wedge	\supset	\equiv	Excl. or	Stroke	Pierce	Inhibit
X	Y								
0	0	0	0	1	1	0	1	1	0
0	1	1	0	1	0	1	1	0	0
1	0	1	0	0	0	1	1	0	1
1	1	1	1	1	1	0	0	0	0

VSBOOL (OP, B, S2,S3)

$S3[i] := B \text{ OP } S2[i]$ where B is a Boolean expression and OP as in VBOOL. B is evaluated only once, on entry to the procedure.

Note: VSBOOL (1,true,V1,V1) sets each element of V1 to true
 VSBOOL (2,false,V1,V1) sets each element of V1 to false
 VSBOOL (2,true, V1,V2) copies V1 into V2

VREL (OP, V1, V2, S3)

compare element by element vectors V1 and V2:

$S3[i] := V1[i] \text{ OP } V2[i]$ where OP is an operation as follows:

$\text{OP} \leq 1 <$
 $= 2 \leq$
 $= 3 =$
 $= 4 \geq$
 $= 5 >$
 $\geq 6 \neq$

Furthermore, VREL takes the value true if and only if every element of the result vector S3 is true.

VSREL (OP,E,V2,S3)

as in VREL but E is a real expression. E is evaluated only once, on entry to the procedure.

The full algorithms for the procedures are given below.

The corresponding procedures are designed in such a way that the length of the result vector has priority over other vector lengths: if necessary, operand vectors are truncated or filled with zeroes or false.

```

procedure VINIT (OP,E,S,V1); value OP,E,S;
  integer OP; real E,S; array V1;
  begin integer I;
    if OP  $\leq 1$  then begin for I:=lwb(V1,1) step 1 until upb (V1,1)
      do V1 [I] :=E
    end
    else
    if OP = 2 then begin V1 [lwb(V1,1)] :=S;
      for I:=lwb(V1,1)+1 step 1 until upb(V1,1)
        do V1 [I]:=V1 [I-1]+E
      end
    else
    if OP  $\geq 3$  then begin V1 [lwb(V1,1)] :=S;
      for I:=lwb(V1,1)+1 step 1 until upb(V1,1)
        do V1 [I] :=V1 [I-1]*E
      end
    end VINIT

```

```

procedure VXMIT (OP,V1,V2,INIT,STEP); value OP, INIT, STEP;
  array V1,V2; integer OP, INIT, STEP;
  begin integer I,J,L1,L2 ;
    L1:= lwb (V1,1); U1:=upb (V1,1)-L1;
    L2:= lwb (V2,1); U2:=upb (V2,1)-L2;
    if OP  $\leq 1$  then
      begin for I:= 0 step 1 until U1 do
        begin J:= INIT+I*STEP;
          if J  $\geq 0 \wedge J \leq U2$  then V2[L2+J] :=V1[L1+I]
        end
      end
    end

```

```

        else for I:=0 step 1 until U1 do
            begin J:= INIT+I*STEP;
                 V1[L1+I] := if J ≥ 0 ∧ J ≤ U2 then V2[L2+J] else 0
            end
end VXMIT;

procedure VMONOD (OP, V1,V2); value OP; integer OP; array V1,V2;
    begin integer L1,U1,L2,U2,L,I;
        switch ACTION:=XMIT, REVERS,XFORM,NEGATE,ABSOLUTE,FLOOR,CEILING,SIGNE,
                DELTA,MEAN;

            L1:=1wb (V1,1); U1:=upb (V1,1)-L1;
            L2:=1wb (V2,1); U2:=upb (V2,1)-L2;
            L:=if U2 ≤ U1 then U2 else U1;
            goto ACTION [if OP ≤ 1 then 1 else if OP ≥ 10 then 10 else OP];

XMIT: OP equals 1:
    for I:=0 step 1 until L do V2[L2+I] :=V1 [L1+I];
    goto FILLING;

REVERS: OP equals 2:
    for I:=0 step 1 until L do V2[L2+I] :=V1 [U1+L1-I];
    goto FILLING;

XFORM: OP equals 3:
    for I:=0 step 1 until L do V2[L2+I] :=entier (V1 [L1+I] +.5);
    goto FILLING;

NEGATE: OP equals 4:
    for I:=0 step 1 until L do V2[L2+I] := -V1 [L1+I];
    goto FILLING;

ABSOLUTE: OP equals 5:
    for I:=0 step 1 until L do V2[L2+I] := abs(V1 [L1+I]);
    goto FILLING;

FLOOR: OP equals 6:
    for I:=0 step 1 until L do V2[L2+I] :=entier (V1 [L1+I]);
    goto FILLING;

CEILING: OP equals 7:
    for I:=0 step 1 until L do V2[L2+I] := if V1 [L1+I] =entier(V1 [L1+I])
                                           then V1 [L1+I] else entier(V1 [L1+I] +1);
    goto FILLING;

SIGNE: OP equals 8:
    for I:=0 step 1 until L do V2[L2+1] :=sign (V1 [L1+I]);
    goto FILLING;

```

DELTA: OP equals 9:
for I:=0 step 1 until L-1 do
 V2[L2+I] := V1[L2+I+1] - V1[L1+I];
 V2[L2+L] := (if L+1 ≤ U1 then V1 [L1+L+1] else 0) - V1 [L1+L];
goto FILLING;

MEAN: OP equals 10:
for I:=0 step 1 until L-1 do
 V2[L2+I] := (V1 [L2+I+1] - V1 [L1+I])/2;
 V2[L2+L] := ((if L+1 ≤ U1 then V1 [L1+L+1] else 0)-V1 [L1+L])/2;

FILLING: Fill V1 with zeroes:
for I:=L+1 step 1 until U2 do V2[L2+I] :=0

end VMONOD;

procedure VDIAD (OP,V1,V2,V3); value OP; integer OP; array V1,V2,V3;

begin integer L1,U1,L2,U2,L3,U3,I;
real procedure action (OP,X,Y); value OP,X,Y;
 integer OP; real X,Y;
 action:= if OP ≤ 1 then X+Y else
 if OP = 2 then X-Y else
 if OP = 3 then X*Y else
 if OP = 4 then X/Y else
 if OP = 5 then (X+Y)/2
 else (X-Y)/2;
 L1:=1wb (V1,1); U1:=upb (V1,1)-L1;
 L2:=1wb (V2,1); U2:=upb (V2,1)-L2;
 L3:=1wb (V3,1); U3:=upb (V3,1)-L3;

 for I:=0 step 1 until U3 do
 V3[L3+I] :=action (OP, if I ≤ U1 then V1 [L1+I] else 0,
 if I ≤ U2 then V2 [L2+I] else 0)

end VDIAD;

procedure VSDIAD (OP,E,V2,V3); value OP,E;
 integer OP; real E; array V2,V3;

begin integer L2,U2,L3,U3,I;
real procedure action (OP,X,Y) value OP,X,Y;
 integer OP; real X,Y;
 action:= if OP ≤ 1 then X+Y else
 if OP = 2 then X-Y else
 if OP = 3 then X*Y
 else X/Y;
 L2:=1wb (V2,1); U2:=upb (V2,1)-L2;
 L3:=1wb (V3,1); U3:=upb (V3,1)-L3;

 for I:=0 step 1 until U3 do
 V3[L3+I] :=action (OP,E, if I ≤ U2 then V2 [L2+I] else 0)

end VSDIAD;

```

real procedure VPROSUM (OP,V1); value OP;
    integer OP, array V1;

    begin integer I, LB; real P;
        LB:=lub(V1,1); P:=V1[LB];
        for I:=LB+1 step 1 until upb (V1,1) do
            P:= if OP ≤ 1 then P+V1[I]
                else P*V1[I];
        VPROSUM:=P
    end VPROSUM;

real procedure VDOT (V1,V2); array V1,V2;

    begin integer L1,U1,L2,U2,L,I;
        real R;
        R:=0;
        L1:=1wb (V1,1); U1:=upb (V1,1)-L1;
        L2:=1wb (V2,1); U2:=upb (V2,1)-L2;
        L:= if U1 ≤ U2 then U1 else U2;
        for I:=0 step 1 until L do R:=R+V1[L1+I] *V2[L2+I];
        VDOT:=R
    end VDOT;

procedure BNOT (S1,S2); boolean array S1,S2;

    begin integer L1,U1,L2,U2,L,I;
        L1:=1wb (S1,1); U1:=upb (S1,1)-L1;
        L2:=1wb (S2,1); U2:=upb (S2,1)-L2;
        L:=if U1 ≤ U2 then U1 else U2;
        for I:=0 step 1 until L do S2[L2+I] :=¬S1[L1+I];
        for I:=L+1 step 1 until U2 do S2[L2+I] :=false
    end BNOT;

procedure VBOOL (OP,S1,S2,S3); value OP;
    integer OP; boolean array S1,S2,S3;

    begin integer L1,U1,L2,U2,L3,U3,I;
        boolean procedure action (OP,X,Y); value OP,X,Y;
            integer OP; boolean X,Y;
            action:=if OP ≤ 1 then X∨Y else
                if OP = 2 then X∧Y else
                if OP = 3 then X⊃Y else
                if OP = 4 then X≡Y else
                if OP = 5 then ¬(X≡Y) else
                if OP = 6 then ¬(X∧Y) else
                if OP = 7 then ¬(X∨Y)
                    else X∧¬Y;
    end

```

```

L1:=1wb (S1,1); U1:=upb (S1,1)-L1;
L2:=1wb (S2,1); U2:=upb (S2,1)-L2;
L3:=1wb (S3,1); U3:=upb (S3,1)-L3;
for I:=0 step 1 until U3 do
    S3[L3+I] :=action (OP, if I ≤ U1 then S1[L1+I] else false,
                      if I ≤ U2 then S2[L2+I] else false)
end VBOOL;

procedure VSBOOL (OP,B,S2,S3); value OP,B; integer OP; boolean B; boolean array S2,S3;
begin integer L2,U2,L3,U3,I;
    boolean procedure action (OP,X,Y); value OP,X,Y;
        integer OP; boolean X,Y;
    action:= if OP ≤ 1 then X∨Y else
              if OP = 2 then X∧Y else
              if OP = 3 then X⊃Y else
              if OP = 4 then X≡Y else
              if OP = 5 then ¬(X≡Y) else
              if OP = 6 then ¬(X∧Y) else
              if OP = 7 then ¬(X∨Y)
              else
                  X∧¬Y;

    L2:=1wb (S2,1); U2:=upb (S2,1)-L2;
    L3:=1wb (S3,1); U3:=upb (S3,1)-L3;
    for I:=0 step 1 until U3 do
        S3[L3+I] :=action (OP,B,if I ≤ U2 then S2[L2+I] else false)
    end VSBOOL;

boolean procedure VREL (OP,V1,V2,S3); value OP; integer OP; array V1,V2; boolean array S3;
begin integer L1,U1,L2,U2,L3,U3,I; boolean B;
    boolean procedure action (OP,X,Y); value OP,X,Y; integer OP; real X,Y;
    action:=if OP ≤ 1 then X < Y else
            if OP = 2 then X ≤ Y else
            if OP = 3 then X = Y else
            if OP = 4 then X ≥ Y else
            if OP = 5 then X > Y
            else
                X ≠ Y;

    B:=true;
    L1:=1wb(V1,1);U1:=upb(V1,1)-L1;
    L2:=1wb(V2,1);U2:=upb(V2,1)-L2;
    L3:=1wb(S3,1);U3:=upb(S3,1)-L3;
    for I:=0 step 1 until U3 do
        begin
            S3[L3+I] :=action (OP, if I ≤ U1 then V1[L1+I] else 0,
                              if I ≤ U2 then V2[L2+I] else 0)
            B:=B∧S3[L3+I];
        end;
    VREL:=B
end VREL;

```

boolean procedure VSREL (OP,E,V2,S3); value OP,E; integer OP; real E; array V2; boolean array S3;

begin integer L2,U2,L3,U3,I; boolean B;

boolean procedure action (OP,X,Y); value OP,X,Y; integer OP; real X,Y;

action:=if OP \leq 1 then X < Y else
if OP = 2 then X \leq Y else
if OP = 3 then X = Y else
if OP = 4 then X \geq Y else
if OP = 5 then X > Y
else X \neq Y;

L2:=lwb(V2,1);U2:=upb(V2,1)-L2;

L3:=lwb(S3,1);U3:=upb(S3,1)-L3;

B:=true;

for I:=0 step 1 until U3 do

begin

S3[L3+I] := action (OP,E if 1 \leq U2 then V2[L2+I] else 0)

B:=B \wedge S3[L3+I];

end;

end VSREL;

Due to the ALGOL 4 implementation of arrays (row by row), the eleven vector procedures may be used by replacing one or more one-dimension arrays by n-dimensional arrays (of the same kind).

Example 1: array M[1:3,2:4]; M[1,2] := 1;

VINIT (2,1,M)

L: . . .

when control reaches the label L, M is filled as follows:

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

Example 2: array V,W[-4:4],M,N [1:3,2:4];

W[-4] := 1;

VINIT (1,1,V); VINIT(2,1,W);

VDIAD(1,V,W,M); VMONOD(2,M,N);

L:

when control reaches the label L, N is filled as follows:

$$\begin{pmatrix} 10 & 9 & 8 \\ 7 & 6 & 5 \\ 4 & 3 & 2 \end{pmatrix}$$

Example 3: array C[1:N,1:N] , T[1:M,1:P] , VM[1:M] , VN[1:N-1] , VP[1:P];

 FETCH UPPER SUB DIAGONAL OF C:
 VXMIT(2,VN,C,2,N+1);
 SEND 0 INTO EACH ELEMENT OF ROW 4 OF T:
 VINIT(1,0,0,VP);
 VXMIT(1,VP,T,4,M);
 ...

GLOSSARY

E

ADVANCED ACCESS METHODS (AAM) – A file manager that processes indexed sequential, direct access, and actual key file organizations and supports the Multiple-Index Processor. See CYBER Record Manager.

BASIC ACCESS METHODS (BAM) – A file manager that processes sequential and word addressable file organizations. See CYBER Record Manager.

BLOCKS – The term block has several meanings depending on context. On tape, a block is information between interrecord gaps on tape. CYBER Record Manager defines several blocks depending on organization:

Organization	Blocks
Index sequential	Data block, index block
Direct access	Home block; overflow block
Actual key	Data block
Sequential	Block type I, C, K, E

A block, in ALGOL, is a compound statement in which one or more declarations precede the first statement.

BOI (Beginning-of-Information) – CYBER Record Manager defines beginning-of-information as the start of the first user record in a file. System-supplied information, such as an index block or control word, does not affect beginning-of-information. Any label on a tape exists prior to beginning-of-information.

CENTRAL MEMORY (CM) – The area of the computer in which all programs reside when they are being executed. A program in central memory has access to the central processor.

CHANNEL – The set of descriptive information by which one reaches or knows of a data file.

CHANNEL NUMBER – The number identifying an input or output device. It is also the name of the channel.

COMPASS – The standard assembly language used with CDC computers.

CONTROL STATEMENT – An instruction to the operating system or its loader. It is found in a section at the beginning of a job deck.

CONTROL WORD – A system-supplied word that precedes each W type record in storage.

CYBER RECORD MANAGER – A generic term relating to the common products BAM and AAM, which run under the NOS and NOS/BE operating systems and allow a variety of record types, blocking types, and file organizations to be created and accessed. The execution time input/output is implemented through CYBER Record Manager for the following products: COBOL 4, COBOL 5, FORTRAN Extended 4, Sort/Merge 4, ALGOL 4, and the DMS-170. Neither the input/output of the NOS and NOS/BE operating systems themselves nor any of the system utilities, such as COPY or SKIPF, are implemented through CYBER Record Manager. All CYBER Record Manager file processing requests ultimately pass through the operating system input/output routines.

DAYFILE - A log the system maintains during execution of a job. All control statements executed by the job, significant information such as file assignment or file disposition, all operator interactions with a job, and errors are recorded in this file. At job termination, it is appended to the file OUTPUT for the job.

DEFAULT - A system-supplied parameter value or name used when a value or name is not supplied by the user.

DELIMITER - One of a set of characters or symbols that is used to separate and organize data items.

DIRECT ACCESS FILE - In the context of CYBER Record Manager, a direct access file is one of the five file organizations. It is characterized by the system hashing of the unique key within each file record to distribute records randomly in blocks called home blocks of the file.

In the context of NOS permanent files, a direct access file is a file that is accessed and modified directly, as contrasted with an indirect access permanent file.

EDIT FILE - Temporary work area that holds created or updated files through the interactive syntax checker ALGEDIT.

EIO (End-of-Information) - CYBER Record Manager defines end-of-information in terms of the file organization and file residence:

File Organization	File Residence	Physical Position
Sequential	Mass storage	After last user record
	Labeled tape in SI, I, X, S, L format	After that user record and before any file trailer labels
	Unlabeled tape in SI, I, X format	After last user record and before any file trailer labels
	Unlabeled tape in S or S format	Undefined
Word Addressable	Mass storage	After last word allocated to file, which might be beyond the last user record
Indexed Sequential, Actual Key	Mass storage	After record with highest key value
Direct Access	Mass storage	After last record in most recently created overflow block or home block with the highest relative address

EXTENDED CORE STORAGE (ECS) - An optional unit or group of units of extra memory storage that enhances the computing capabilities of the computer by providing transfer of information to and from central memory at very high speed.

FIELD LENGTH (FL) - The number of main memory words assigned to a job.

FILE - A collection of records treated as a unit.

HARDWARE LANGUAGE - A machine dependent language that uses ALGOL symbols that the computer can recognize; it is the language used by the programmer. The hardware representation of ALGOL symbols are shown in Appendix C.

HOME BLOCK - Mass storage allocated for a file with direct access organization at the time the file is created.

INDEXED FILE - A file organization in which records are stored in ascending order by key.

LARGE CORE MEMORY (LCM) - Holds instructions and operands for fast transfer to or from SCM and operands to or from the operating registers if it is available.

LEVEL - For system-logical-records, an octal number 0 through 17 in the system-supplied 48-bit marker that terminates a short or zero-length PRU.

LGO - The default name of the file to which language processors write executable code during program assembly or compilation.

LOGICAL RECORD - Under NOS, a data grouping that consists of one or more PRUs terminated by a short PRU or zero length PRU. Equivalent to a system-logical-record under NOS/BE.

MACRO - The statement that calls the macro function is replaced by the machine code necessary to perform the function during assembly.

MAIN OVERLAY - An overlay that must remain in memory throughout execution of an overlaid program.

NOISE RECORD - Number of characters the tape drivers discard as being extraneous noise rather than a valid record. Value depends on installation settings.

OVERFLOW BLOCK - Mass storage the system adds to a file with direct access organization when records cannot be accommodated in the home block.

OVERLAY - A technique for bringing routines into main memory from storage during processing so several routines occupy the same memory locations at different times. Overlay is used when the total memory requirements for instructions exceed the available main memory.

PARTITION - CYBER Record Manager defines a partition as a division within a file with sequential organization. Generally, a partition contains several records or sections. Implementation of a partition boundary is affected by file structure and residence.

Device	RT	BT	Physical Boundary
PRU	W	I	A short PRU of level 0 containing one-word deleted record pointing back to last I block boundary, followed by a control word with flag indicating partition boundary.
	W	C	A short PRU of level 0 containing a control word with a flag indicating partition boundary.
	D, F, R, T, U, Z	C	A short PRU of level 0 followed by a zero-length PRU of level 17.
S or L format type	W	I	Separate tape block containing as many deleted records of record length 0 as required to exceed noise record size, followed by a deleted one-word record pointing back to the last I block boundary, followed by a control word with flag indicating a partition boundary.
	W	C	Separate tape block containing as many deleted records or record length 0 as required to exceed noise record size, followed by a control word with a flag indicating a partition boundary.
	D, F, T, R, U, Z	C, K, E	Tapemark
	S	—	Zero-length PRU of level number 0.
Any other tape format			Undefined

Notice that in a file with W type records, a short PRU of level 0 terminates both a section and a partition.

PRIMARY OVERLAY - A second level overlay that is subordinate to the main overlay.

PROGRAM - A block or compound statement which is not contained within another statement and which makes no use of other statements not contained within it.

PROGRAM DEPTH - The number of nested procedure declarations within which a block is contained.

PROGRAM LEVEL - The largest set of blocks of the same depth, such that one block of the set encloses all of the other blocks (if any) of the set.

PRU - Under NOS and NOS/BE, the amount of information transmitted by a single physical operation of a specified device. The size of a PRU depends on the device:

Device	Size in Number of 60-Bit Words
Mass storage	64
Tape in SI format with coded data	128
Tape in SI, X, or I format with binary data	512
Tape in other format	undefined

A PRU which is not full of user data is called a short PRU; a PRU that has a level terminator but no user data is called a zero-length PRU.

PRU DEVICE - Under NOS and NOS/BE, a mass storage device or tape in SI, I, or X format, so called because records on these devices are written in PRUs.

RANDOM FILE - In the context of CYBER Record Manager, a file with word addressable, indexed sequential, direct access, or actual key organization in which individual records can be accessed by the value of their keys.

In the context of the NOS or NOS/BE operating systems, a file with the random bit set in the file environment table in which individual records are accessed by their relative PRU numbers.

RECORD - CYBER Record Manager defines a record as a group of related characters. A record or a portion thereof is the smallest collection of information passed between CYBER Record Manager and a user program. Eight different record types exist, as defined by the RT field of the file information table.

Other parts of the operating systems and their products might have additional or different definitions of records.

REFERENCE LANGUAGE - A machine independent language that uses a set of basic symbols to define the syntax and semantics of the ALGOL language.

RUN-TIME SYSTEM - A routine that controls the execution of an ALGOL object program. It is external to the generated object code.

SCALAR SPACE - Includes storage for all those items which can be quantified at compile time, such as simple variables declared in the block, array descriptors, dope vectors, and various compiler generated locations. It does not include storage for arrays.

SECONDARY OVERLAY - The third level of overlays. A secondary overlay is called into memory by its associated primary overlay.

SECTION - CYBER Record Manager defines a section as a division within a file with sequential organization. Generally, a section contains more than one record and is a division within a partition of a file. A section terminates with a physical representation of a section boundary:

Device	RT	BT	Physical Representation
PRU device	W	I	Deleted one-word record pointing back to last I block boundary followed by a control word with flags indicating a section boundary. At least the control word is in a short PRU of level 0.
	W	C	Control word with flags indicating a section boundary. The control word is in a short PRU of level 0.
	D, F, R, T, U, Z	C	Short PRU with level less than 17 octal.
	S	any	Undefined
S or L format tape	W	I	A separate tape block containing as many deleted records of record length 0 required to exceed noise record size followed by a deleted one-word record pointing back to last I block boundary followed by a control word with flags indicating a section boundary.
	W	C	A separate tape block containing as many deleted records of record length 0 required to exceed noise record size followed by a control word with flags indicating a section boundary.
	D, F, R, T, U, Z	C, K, E	Undefined
	S	any	Undefined
Any other tape format			Undefined

The NOS 1 and NOS/BE 1 operating systems equate a section with a system-logical record of level 0 through 16 octal.

SEQUENTIAL FILE - Records are placed in physical order rather than in logical order. Given the location of one record, the location of the next record is determined in relation to the given record only.

SMALL CORE MEMORY - Holds operands and executable instructions immediately prior to placement in the instruction stack for execution.

SOURCE DECK - Consists of the source lines constituting one source program or one source procedure.

SOURCE LINE - The equivalent of a card or of a card image.

SOURCE LISTING - A printed listing of a compiled source program or source procedure. Each line in the listing corresponds to one source line.

SYMBOLIC DUMP - A listing of current values of declared variables that is produced when an executing program terminates due to an error. The D option must be selected at compile time and run time for a dump to be printed.

VECTOR - A one dimensional array.

WORD ADDRESSABLE FILE - A collection of 60-bit words on a mass storage device. Each word has an ordinal, called a word address, associated with its physical placement in the file. By stating a word address, the contents of that word can be accessed.

INDEX

A
Parameters I, L, R, A, N, O on ALGOL Control Statement 6-2
Abnormal Termination
Object Time Abnormal Termination Dump 13-1
Abort
Compiler Abort 15-3
AIDA ABORT Command to Terminate Execution 16-11
Actual
Definition of Procedure Identifier, Actual Parameter, Function Designator 2-17
Correspondence between Actual Parameter and Formal Parameter 2-40
ADD
ALGEDIT ADD Command 17-10
ADDRESS
Macro ADDRESS B-8
Advisory Messages
Advisory Messages 15-3
AIDA
ALGOL Interactive Debugging Aids (AIDA) 16-1
Alarms
Compiler Alarms 15-1
ALGEDIT
ALGEDIT Diagnostics 15-4
ALGEDIT Interactive Syntax Checker 17-1
ALGEDIT. Command 17-4
ALGOL
ALGOL Compiler Features 1-1
Additional Delimiters Code, Algol, Fortran 2-10
Code, Algol Fortran for Separate Compilation of Procedure 2-41
Use of Symbols Code, Algol, Fortran 2-50
ALGOL Control Card Statement Syntax 6-1
Parameters U, C, S on ALGOL Control Statement 6-3
Parameters P, E, B, D, F, K, X on ALGOL Control Statement 6-4
Macro ALGOL B-1
Character Representation of ALGOL Symbols C-1
Parameters I, L, R, A, N, O on ALGOL Control Statement 6-2
ALGOL-60
Contents of Revised Report on ALGOL-60 2-3
Introduction to Revised Report on ALGOL-60 2-4
Index of Revised Report on ALGOL-60 2-57
Input/Output Proposal for ALGOL-60 3-2
Revised Report on ALGOL-60 2-2
ALGORUN
Run Time Supervisory Program ALGORUN 14-1
Alignment Mark
Semantics of Title Format, Alignment Mark 3-11
Alpha Format
Definition of String Format, Alpha Format, Boolean Format 3-7
Semantics of String Format, Alpha Format 3-9

- ANALYZE
 - ALGEDIT ANALYZE Command 17-25
- Arithmetic Operator
 - Definition of Delimiters, Arithmetic Operator, Relational Operator 2-9
 - Definition of Arithmetic Expression, Primary, Factor, Term 2-19
 - Description of Arithmetic Operator 2-20
 - Type of Operand in Arithmetic Expression 2-22
 - Precedence of Arithmetic Operator 2-23
 - Arithmetic Expression in a For List 2-36
- Array
 - Array Subscript Bounds 2-44
 - Array Dimension 2-44
 - Array Type 2-45
 - Virtual Array 2-45
 - Array and Subscripted Variable Monitoring 9-7
 - Virtual Array 11-1
- Array Declaration
 - Definition of Array Declaration 2-44
- ARTHOFLW
 - Control Procedures ARTHOFLW, PARITY, EOF 3-49
- Assembly Language
 - Assembly Language Object Code 5-1
- ASSIGN
 - Macro ASSIGN B-7
- Assignment Statement
 - Definition of Assignment Statement 2-30
 - Evaluation of Expression in Assignment Statement 2-30
 - Definition of Type of Variable in Assignment Statement 2-31
- B
 - Parameters P, E, B, D, F, K, X on ALGOL Control Statement 6-4
 - Parameters C, B, I, W on CHANNEL Statement 7-1
- BACKSPACE
 - Hardware Function Procedures UNLOAD, BACKSPACE 3-51
- Backus Normal Form
 - BNF (Backus Normal Form) Syntax Description 2-7
- BAD DATA
 - Control Procedures BAD DATA, ERROR 3-49
- Basic Symbol
 - Definition of Basic Symbol 2-8
- Batch
 - Use of AIDA in Batch Mode 16-13
- Binary
 - Binary Output from Compilation 5-1
- Block
 - Definition of Statement, Block Head, Block 2-27
 - Description of Block and Program 2-29
- Block Head
 - Definition of Statement, Block Head, Block 2-27
- BNF
 - BNF (Backus Normal Form) Syntax Description 2-7
- BNOT
 - Text of Procedures BNOT, VBOOL D-7
 - Procedures VDOT, BNOT, VBOOL, VSBOOL D-3
- Body Replacement
 - Body Replacement and Execution of Procedure 2-39
- Boolean
 - Variables of Type Boolean 2-24
 - Data Type Real, Integer, Boolean 2-43

- Boolean Expression
 - Definition of Boolean Expression 2-24
 - Definition of String Format, Alpha Format, Boolean Format 3-7
 - Semantics of Non-format, Boolean Format 3-10
- Boolean Operator
 - Truth Table for Boolean Operator 2-25
 - Precedence of Boolean Operator 2-25
- Bounds
 - Array Subscript Bounds 2-44
- Bracket
 - Definition of Bracket, Declarator, Specificator 2-10
- Brackets
 - <and > as Metalinguistic Brackets 2-8
- BREAK
 - BREAK Command to Set Breakpoints in AIDA Execution 16-6
- Breakpoints
 - Releasing Breakpoints in AIDA Execution 16-9
 - Setting Breakpoints in AIDA Execution 16-6
- BYE
 - ALGEDIT BYE Command 17-5
- C
 - Parameters U, C, S on ALGOL Control Statement 6-3
 - Parameters C, B, I, W on CHANNEL Statement 7-1
 - Parameters C, P, T on LGO Statement 8-2
- Call-by-Name
 - Call-by-Value and Call-by-Name for Procedure 2-39
- Call-by-Value
 - Call-by-Value and Call-by-Name for Procedure 2-39
- CHANERROR
 - Control Procedures CHANERROR 3-50
 - Error Control Procedures ERROR and CHANERROR 15-4
- CHANNEL
 - Parameters C, B, I, W on CHANNEL Statement 7-1
 - Channel Equate Statement 7-2
 - Serial File Parameters on CHANNEL Statement 7-1
- Channel Statements
 - Channel Statements 7-1
 - Standard ALGOL Channel Statements 7-3
- Channel Number
 - Unit Number and Channel Number 3-16
- Character
 - Character Representation of ALGOL Symbols C-1
 - Standard Character Sets A-1
 - Meaning of Special Characters in ALGEDIT 17-3
- CHECK
 - ALGEDIT CHECK Command 17-24
- Checkoff
 - Additional Delimiters Overlay, Virtual, Checkon, Checkoff 2-10
- Checkon
 - Additional Delimiters Overlay, Virtual, Checkon, Checkoff 2-10
- CHLENGTH
 - Procedures CHLENGTH, IN REAL, OUT REAL 3-26
 - Procedures CHLENGTH, STRING ELEMENT 3-45

- CLOCK
 - Procedures CLOCK, THRESHOLD, INRANGE 3-52
- COBOL
 - COBOL Format for ALGEDIT Lines 17-7
 - Compilation and Execution of a COBOL Program through ALGEDIT 17-26
- Code
 - Additional Delimiters Code, Algol, Fortran 2-10
 - Code, Algol, Fortran for Separate Compilation of Procedure 2-41
 - Definition of Code 2-47
 - Code as Procedure Body 2-50
 - Use of Symbols Code, Algol, Fortran 2-50
 - Example of Use of Code 2-51
 - Use of Code in Source Procedure 4-3
- Comma
 - Suppression of Sign, Zero, Comma 3-5
- Commands
 - Interactive Commands for AIDA 16-3
 - ALGEDIT Command Syntax 17-2
- Command Mode
 - ALGEDIT Command Mode 17-4
- Comments
 - Definition of Comment 2-10
 - Comments in AIDA 16-12
- COMPASS
 - COMPASS Format for ALGEDIT Lines 17-8
 - Assembly and Execution of a COMPASS Program through ALGEDIT 17-26
- Compilation
 - Input to Compilation 4-1
 - Output from Compilation 5-1
 - Source Listing Produced by Compilation 5-2
 - Program Compilation through ALGEDIT 17-26
- Compiler
 - ALGOL Compiler Features 1-1
 - Compiler Output 1-2
 - Routines Included in Compiler 1-2
 - Compiler Structure 1-2
 - Compiler Diagnostics 15-1
 - Compiler Abort 15-3
- Compiler Stop
 - Compiler Stop 15-3
- Compile Time
 - Compile Time Error Detection 1-1
- Compound Statement
 - Definition of Compound Statement, Program 2-28
- Conditional Statement
 - Definition of Conditional Statement 2-33
- CONIN
 - AIDA CONIN Command to Connect Input File 16-12
- Connect
 - AIDA Commands to Connect Files 16-12
- CONOUT
 - AIDA CONOUT Command to Connect OUTALG File 16-12
- Control Statement
 - ALGOL Control Statement Syntax 6-1
 - Parameters U, C, S, on ALGOL Control Statement 6-3
 - Parameters P, E, B, D, F, K, X on ALGOL Control Statement 6-4
 - Parameters I, L, R, A, N, O on ALGOL Control Statement 6-2
 - Operating System Control Statement Entered through ALGEDIT 17-1

- Control Procedures
 - Control Procedures ARTHOFLW, PARITY, EOF 3-49
 - Control Procedures BAD DATA, ERROR 3-49
 - Control Procedures CHANERROR 3-50
- Controlled
 - Controlled Variable in a For Clause 2-37
- CREATE
 - ALGEDIT CREATE Command 17-9
- Cross Reference
 - Cross Reference Listing 13-2
- D
 - Parameters P, E, B, D, F, K, X on ALGOL Control Statement 6-4
 - Parameters S and D on LGO Statement 8-1
- Debugging
 - Debugging Facilities 9-1
 - Debugging Directives Trace, Snap, Snapoff 9-1
 - Debugging Output and Label Monitoring 9-5
- Debugging Aids
 - ALGOL Interactive Debugging Aids (AIDA) 16-1
- Decimal Number
 - Definition of Integer, Decimal Number, Number 2-12
- Decimal Point
 - Decimal Point on Input/Output 3-5
- Declaration
 - Definition of Declaration 2-41
- Declarator
 - Definition of Bracket, Declarator, Specificator 2-10
- DELETE
 - ALGEDIT DELETE Command 17-12
- Delimiters
 - Definition of Delimiters, Arithmetic Operator, Relational Operator 2-9
 - Additional Delimiters Code, Algol, Fortran 2-10
 - Additional Delimiters Trace, Snap, Snapoff 2-10
 - Additional Delimiters Overlay, Virtual, Checkon, Checkoff 2-10
- Depth
 - Program Depth and Program Level 14-2
- Designational Expression
 - Definition of Designational Expression, Label, Switch 2-26
- Diagnostics
 - Diagnostics in Source Listing 5-2
 - Compiler Diagnostics 15-1
 - Object Time Diagnostics 15-4
- ALGEDIT Diagnostics 15-4
- Digit
 - Definition of Digit 2-9
- Dimension
 - Array Dimension 2-44
- Direct Access
 - Direct Access Procedures 3-46
- Disconnect
 - AIDA DISOUT Command to Disconnect OUTALG File 16-12

- DISOUT
 - AIDA DISOUT Command to Disconnect OUTALG File 16-12
- Display
 - Display Formats in AIDA 16-2
- Dummy Statement
 - Definition of Dummy Statement 2-32
- DUMP
 - Procedures IOLTH, POSITION, DUMP 3-51
 - Structured Dump 13-1
 - Dump File Diagnostics 15-4
 - Object Time Abnormal Termination Dump 13-1
- E
 - Parameters P, E, B, D, F, K, X on ALGOL Control Statement 6-4
- ECS
 - ECS, LCM Procedures READ ECS, WRITE ECS 3-56
 - Use of LCM and ECS 11-1
 - LCM, SCM, and ECS Used at Object Time 14-2
- EDIT
 - ALGEDIT EDIT Command 17-17
- Else
 - Meaning of If, Then, Else 2-33
- END
 - AIDA END Command to Terminate Execution 16-11
 - ALGEDIT END Command 17-5
- ENDFILE
 - Hardware Function Procedures ENDFILE, REWIND 3-50
- End-of-Partition
 - End-of-Partition 3-54
- Entier
 - Transfer Function such as Entier 2-19
- EOF
 - Control Procedures ARTHOFLW, PARITY, EOF 3-49
- EOP
 - Symbol EOP 4-1
- Error
 - Compile Time Error Detection 1-1
 - Control Procedures BAD DATA, ERROR 3-49
 - Input and Output Error Processing 3-53
 - Transmission Error Processing 3-54
 - Error Control Procedures ERROR and CHANERROR 15-4
 - Macro ERROR B-11
- Errors
 - AIDA Fatal Errors 16-13
- Error Messages
 - ALGEDIT Error Messages 15-4
- Evaluation
 - Side-Effect during Evaluation of Function Designator 2-17

Example

- Example of Procedure 2-48
- Example of Use of Code 2-51
- Example of Procedure Declaration 2-55
- Example of Number Format 3-4
- Example of Layout Procedure 3-22
- Example of Use of Input/Output Procedures 3-28
- Pascals Triangle Example of Input/Output 3-43
- Example of Overlay Use 10-3
- Example of Virtual Array 11-1
- Example of Object Time Stack 14-4
- Example of Macro Usage B-14
- Example Programs C-2, C-3
- Example of Vector Procedure Use D-9

Execution

- Object Time Program Execution 1-2
- Body Replacement and Execution of Procedure 2-39
- Execution Time Options 8-1
- Object Time Execution 14-1
- Run Time Execution 14-1
- Program Compilation and Execution through ALGEDIT 17-26

Exponent Part

- Exponent Part on Input/Output 3-6

Expression

- Definition of Expression, Simple Variable 2-15
- Evaluation of Expression in Assignment Statement 2-30

Extended Core Storage

- Extended Core Storage, Large Core Memory Procedures READ ECS, WRITE ECS 3-56
- Use of Large Core Memory and Extended Core Storage 11-1

External Identifier

- Definition of External Identifier 2-11

F

- Parameters P, E, B, D, F, K, X on ALGOL Control Statement 6-4

Factor

- Definition of Arithmetic Expression, Primary, Factor, Term 2-19

False

- Logical Values True and False 2-9

Fatal Errors

- AIDA Fatal Errors 16-13

FETCHITEM

- Procedures FETCHITEM, STOREITEM 3-48

FETCHLIST

- Procedures FETCHLIST, STORELIST 3-48

File

- File Creation and Modification through ALGEDIT 17-6

For

- Meaning of For 2-35

For Clause

- Controlled Variable in a For Clause 2-37

For List

- For List Elements 2-36
- Arithmetic Expression in a For List 2-36
- Step Until Element in a For List 2-37
- While Element in a For List 2-37

For Statement

- Definition of For Statement 2-35
- Go to into a For Statement 2-38

- Formal
 - Correspondence between Actual Parameter and Formal Parameter 2-40
 - Definition of Formal Parameter 2-46
- Format
 - Example of Number Format 3-4
 - Use of Format on Input 3-6
 - Definition of Non-Numeric Format 3-7
 - Summary of Format Codes 3-12
 - Procedure FORMAT 3-20
 - ALGEDIT FORMAT Command 17-6
- Formats
 - Hints for Efficient Use of Formats 3-54
- Formatted
 - Optimization of Simple Formatted Input and Output 12-1
- Format String
 - Definition of Format String 3-11
- FORTRAN
 - Additional Delimiters Code, Algol, Fortran 2-10
 - Code, Algol, Fortran for Separate Compilation of Procedure 2-41
 - Use of Symbols Code, Algol, Fortran 2-50
 - FORTRAN Format for ALGEDIT Lines 17-8
- FORTRAN Extended
 - Compilation and Execution of a FORTRAN Extended Program through ALGEDIT 17-26
- FTN
 - Compilation and Execution of a FORTRAN Extended Program through ALGEDIT 17-26
- Function Designator
 - Definition of Procedure Identifier, Actual Parameter, Function Designator 2-17
 - Side-Effect during Evaluation of Function Designator 2-17
 - Value of Function Designator 2-49
- GET
 - Procedures PUT, GET 3-40
 - Procedures GET, PUT 3-48
- GETARRAY
 - Procedures GETARRAY, PUTARRAY 3-48
- GETITEM
 - Procedures GETITEM, PUTITEM 3-47
- GETLIST
 - Procedures GETLIST, PUTLIST 3-46
- GETSCM
 - Macro GETSCM B-13
- GO
 - AIDA GO Command to Resume Execution 16-11
- GOTO
 - Macro GOTO B-10
- Go to
 - Definition of Go to Statement 2-31
 - Go to into a For Statement 2-38
- Hardware Function Procedures
 - Hardware Function Procedures SKIPF, SKIPB 3-50
 - Hardware Function Procedures ENDFILE, REWIND 3-50
 - Hardware Function Procedures UNLOAD, BACKSPACE 3-51
 - Reference Language, Hardware Language and Publication Language, 2-1, 2-6

- Hardware Language Representation 4-3
- Hints
 - Hints for Efficient Use of Formats 3-54
- HISTORY
 - Use of File HISTORY by AIDA 16-1
- Horizontal
 - Horizontal and Vertical Control 3-17
- H END
 - Procedures H LIM, V LIM, H END, V END 3-21
- H LIM
 - Procedures H LIM, V LIM, H END, V END 3-21
- H SKIP
 - Procedures H SKIP, VSKIP 3-39
- I
 - Parameters C, B, I, W on CHANNEL Statement 7-1
 - Parameters I, L, R, A, N, O on ALGOL Control Statement 6-2
- Identifier
 - Definition of Identifier 2-11
 - Standard Procedures and Reserved Identifier 2-18
- If
 - Meaning of If, Then, Else 2-33
- If Statement
 - Definition of If Statement 2-32
- Implicit Outer Block Head
 - Implicit Outer Block Head 4-1
- INALG
 - AIDA CONIN Command Connects INALG 16-12
 - Use of File INALG by AIDA 16-1
- INARRAY
 - Procedures INARRAY, OUTARRAY 3-27
- INCHARACTER
 - Procedures INCHARACTER, OUTCHARACTER 3-25
- Index
 - Index of Revised Report on ALGOL 60 2-57
- Indexed
 - Indexed List Input and Output 3-46
 - Indexed Item Input and Output 3-47
 - Indexed File and Word Addressable Parameters on CHANNEL Statement 7-1
- Input
 - Use of Format on Input 3-6
 - Input and Output Procedure 3-13
 - Input and Output Procedures 3-25
 - Input Procedures 3-35
 - Procedures INPUT, IN LIST 3-35
 - List of Other Input and Output Procedures 3-42
 - Additional Input and Output Procedures 3-45
 - Indexed List Input and Output 3-46
 - Indexed Item Input and Output 3-47
 - Input and Output Error Processing 3-53
 - Use of Files INPUT and OUTPUT in ALGEDIT 17-26
 - Use of File INPUT by AIDA 16-1
 - Optimization of Simple Formatted Input and Output 12-1
- Input/Output
 - Input/Output Proposal for ALGOL-60 3-2
- INRANGE
 - Procedures CLOCK, THRESHOLD, INRANGE 3-52

- Insertion
 - Meaning of Insertion 3-5
 - Definition of Number Format, Replicator, Insertion Sequence 3-3
- Integer
 - Definition of Integer, Decimal Number, Number 2-12
 - Integer and Real Type 2-13
 - Unsigned Integer Label 2-27
 - Data Type Real, Integer, Boolean 2-43
- Interactive
 - ALGEDIT Interactive Syntax Checker 17-1
 - ALGOL Interactive Debugging Aids (AIDA) 16-1
- Interface Macros
 - Interface Macros B-1
- Interrupt
 - Use of Interrupt to Suspend Execution in ALGEDIT 17-27
- IN CONTROL
 - Procedures OUT CONTROL, IN CONTROL 3-40
- IN LIST
 - Procedures INPUT, IN LIST 3-35
 - Algorithm for Executing Procedure IN LIST 3-36
- IN REAL
 - Procedures CHLENGTH, IN REAL, OUT REAL 3-26
- IOLTH
 - Procedures IOLTH, POSITION, DUMP 3-51
- Item
 - Indexed Item Input and Output 3-47

- K
 - Parameters P, E, B, D, F, K, X on ALGOL Control Statement 6-4
- KIND
 - Macro KIND B-3
- Kinds
 - Quantities, Kinds, Scopes, Values, Types 2-15

- L
 - Parameters I, L, R, A, N, O on ALGOL Control Statement 6-2
- Label
 - Definition of Designational Expression, Label, Switch 2-26
 - Unsigned Integer Label 2-27
 - Debugging Output and Label Monitoring 9-5
- Large Core Memory
 - Extended Core Storage, Large Core Memory Procedures READ ECS, WRITE ECS 3-56
 - Use of Large Core Memory and Extended Core Storage 11-1
- LAYOUT
 - Procedure LAYOUT and LIST 3-15
 - Layout Procedures 3-18
 - Example of Layout Procedure 3-22
- LCM
 - ECS, LCM Procedures READ ECS, WRITE ECS 3-56
 - Use of LCM and ECS 11-1
 - LCM, SCM, and ECS Used at Object Time 14-2
- LENGTH
 - Macro LENGTH B-8
- Letter
 - Definition of Letter 2-8
 - Upper Case and Lower Case Letter 2-9

- Level
 - Program Depth and Program Level 14-2
- LGO Statement
 - Parameters S and D on LGO Statement 8-1
 - Parameters C, P, T on LGO Statement 8-2
- Library
 - Library Subprograms 1-3
 - Library Subprograms 14-1
- Line=Text
 - ALGEDIT Line=Text Command 17-11
- LIST
 - Procedure LAYOUT and LIST 3-15
 - List Procedures 3-23
 - Indexed List Input and Output 3-46
 - Procedure LAYOUT and LIST 3-15
 - List Procedures 3-23
 - Indexed List Input and Output 3-46
 - ALGEDIT LIST Command 17-21
- Listing
 - Source Listing Produced by Compilation 5-1
- Logical Operator
 - Definition of Logical Operator, Sequential Operator, Separator 2-9
- Logical Value
 - Definition of Logical Value 2-9
- Lower Bound
 - Definition of Lower Bound, Upper Bound 2-44
- Lower Case
 - Upper Case and Lower Case Letter 2-9
- Macro
 - Macro ALGOL B-1
 - Macro RETURN B-2
 - Macro PARAMS B-3
 - Macro KIND B-3
 - Macro TYPE B-4
 - Macro SPEC B-5
 - Macro VALUE B-6
 - Macro ASSIGN B-7
 - Macro ADDRESS B-8
 - Macro LENGTH B-8
 - Macro ORDER B-9
 - Macro GOTO B-10
 - Macro XFORM B-10
 - Macro STRING B-11
 - Macro ERROR B-11
 - Macro XEQ B-12
 - Macro GETSCM B-13
 - Example of Macro Usage B-14
- MATMULT
 - Procedure MATMULT D-1
- Matrix
 - Standard Procedures for Vector and Matrix Manipulation D-1

- Merge
 - SAVE with Merge in ALGEDIT 17-20
- Metalinguistic
 - Metalinguistic Entities 2-7
- MOVE
 - Procedure MOVE 3-53
- N
 - Parameters I, L, R, A, N, O on ALGOL Control Statement 6-2
- Non-format
 - Semantics of Non-format, Boolean Format 3-10
- Non-Numeric
 - Definition of Non-Numeric Format 3-7
- NO DATA
 - Procedures TABULATION, NO DATA 3-21
- Number
 - Definition of Integer, Decimal Number, Number 2-12
 - Format of a Number, 2-12, 2-13
 - Example of Number Format 3-4
- Number Format
 - Definition of Number Format, Replicator, Insertion Sequence 3-3
- O
 - Parameters I, L, R, A, N, O on ALGOL Control Statement 6-2
- Object Code
 - Assembly Language Object Code 5-1
 - Object Code Structure 14-1
- Object Time
 - Object Time Program Execution 1-2
 - Object Time Error Detection 1-2
 - Object Time Execution 14-1
 - Object Time Stack 14-2
 - LCM, SCM, and ECS Used at Object Time 14-2
 - Example of Object Time Stack 14-4
 - Format of Entities in Object Time Stack 14-5
 - Object Time Diagnostics 15-4
- Object Time Abnormal Termination Dump 13-1
- Operating System
 - Operating System Control Statements Entered through ALGEDIT 17-1
- Optimization
 - Optimization Facilities 1-2
 - Optimization 12-1
- ORDER
 - Macro ORDER B-9
- OUTALG
 - AIDA Commands to Connect and Disconnect OUTALG 16-12
 - Use of File OUTALG by AIDA 16-1
- OUTARRAY
 - Procedures INARRAY, OUTARRAY 3-27
- OUTCHARACTER
 - Procedures INCHARACTER, OUTCHARACTER 3-25

Output

- Input and Output Procedure 3-13
- Input and Output Procedures 3-25
- Output Procedures 3-29
- Procedures OUTPUT, OUT LIST 3-29
- List of Other Input and Output Procedures 3-42
- Additional Input and Output Procedures 3-45
- Indexed List Input and Output 3-46
- Indexed Item Input and Output 3-47
- Input and Output Error Processing 3-53
- Optimization of Simple Formatted Input and Output 12-1
- Use of File OUTPUT by AIDA 16-1
- Use of Files INPUT and OUTPUT in ALGEDIT 17-26

OUT CONTROL

- Procedures OUT CONTROL, IN CONTROL 3-40

OUT LIST

- Procedures OUTPUT, OUT LIST 3-29
- Algorithm for Executing Procedure OUT LIST 3-31

OUT REAL

- Procedures CHLENGTH, IN REAL, OUT REAL 3-26

Overlay

- Additional Delimiters Overlay, Virtual, Checkon, Checkoff 2-10
- Example of Overlay Use 10-3

Overlay Declaration

- Overlay Declaration Semantics 10-1

Own

- Own Declaration 2-42
- Own Variables at Object Time 14-3

P

- Parameters P, E, B, D, F, K, X on ALGOL Control Statement 6-4
- Parameters C, P, T on LGO Statement 8-2

Parameter

- Definition of Procedure Identifier, Actual Parameter, Function Designator 2-17
- Correspondence between Actual Parameter and Formal Parameter 2-40
- Definition of Formal Parameter 2-46
- Parameter Delimiters 2-41

PARAMS

- Macro PARAMS B-3

PARITY

- Control Procedures ARTHOFLW, PARITY, EOF 3-49

Pascals Triangle

- Pascals Triangle Example of Input/Output 3-43

POSITION

- Procedures IOLTH, POSITION, DUMP 3-51

Precedence

- Precedence of Arithmetic Operator 2-23
- Precedence of Boolean Operator 2-25

Primary

- Definition of Arithmetic Expression, Primary, Factor, Term 2-19

Procedure

- Call-by-Value and Call-by-Name for Procedure 2-39
- Body Replacement and Execution of Procedure 2-39
- Code, Algol, Fortran for Separate Compilation of Procedure 2-41
- Example of Procedure 2-48
- Specification in a Procedure 2-49
- Input and Output Procedure 3-13
- Procedure LAYOUT and LIST 3-15
- Procedure and Simple Variable Monitoring 9-6
- Procedure Optimization 12-1
- Standard Procedures and Reserved Identifier 2-18
- Layout Procedures 3-18
- List Procedures 3-23
- Input and Output Procedures 3-25
- Output Procedures 3-29
- Input Procedures 3-35
- List of Other Input and Output Procedures 3-42
- Additional Input and Output Procedures 3-45
- Definition of Procedure Identifier, Actual Parameter, Function Designator 2-17
- Definition of Procedure Statement 2-38
- Definition of Procedure Body 2-47
- Definition of Procedure Declaration 2-47
- Code as Procedure Body 2-50
- Example of Procedure Declaration 2-55

Program

- Definition of Compound Statement, Program 2-28
- Source Input for ALGOL Program 4-1
- Description of Block and Program 2-29

Publication Language

- Reference Language, Hardware Language and Publication Language, 2-1, 2-6

PUT

- Procedures PUT, GET 3-40
- Procedures GET, PUT 3-48

PUTARRAY

- Procedures GETARRAY, PUTARRAY 3-48

PUTITEM

- Procedures GETITEM, PUTITEM 3-47

PUTLIST

- Procedures GETLIST, PUTLIST 3-46

Q

- Q Option Required for AIDA Use 16-1

QFILE

- QFILE as Used by AIDA 16-1

Quantities

- Quantities, Kinds, Scopes, Values, Types 2-15

Quotes

- Hardware Representation of String Quotes 2-14

R

- Parameters I, L, R, A, N, O on ALGOL Control Statement 6-2

READ ECS

- Extended Core Storage, Large Core Memory Procedures READ ECS, WRITE ECS 3-56

- Real
 - Integer and Real Type 2-13
 - Data Type Real, Integer, Boolean 2-43
 - Truncation and Rounding Real Number on Input/Output 3-5
- RECHECK
 - RECHECK Command in ALGEDIT 17-8
- Recovery Messages
 - Recovery Messages 15-2
- Reference Language
 - Reference Language, Hardware Language and Publication Language, 2-1, 2-6
 - Description of Reference Language 2-7
- Relational Operator
 - Definition of Delimiters, Arithmetic Operator, Relational Operator 2-9
 - Definition of Relational Operator 2-23
- RELEASE
 - AIDA RELEASE Command to Release Breakpoints 16-9
- Replicator
 - Definition of Number Format, Replicator, Insertion Sequence 3-3
 - Meaning of Replicator 3-4
- RESEQ
 - ALGEDIT RESEQ Command 17-14
- RETURN
 - Macro RETURN B-2
- Revised Report
 - Contents of Revised Report on ALGOL-60 2-3
 - Introduction to Revised Report on ALGOL-60 2-4
 - Index of Revised Report on ALGOL-60 2-57
 - Revised Report on ALGOL-60 2-2
- REWIND
 - Hardware Function Procedures ENDFILE, REWIND 3-50
- Rounding
 - Truncation and Rounding Real Number on Input/Output 3-5
- RUN
 - ALGEDIT RUN Command 17-26

- S
 - Parameters U, C, S on ALGOL Control Statement 6-3
 - Parameters S and D on LGO Statement 8-1
- SAVE
 - ALGEDIT SAVE Command 17-19
- SCM
 - LCM, SCM, and ECS Used at Object Time 14-2
- Scopes
 - Quantities, Kinds, Scopes, Values, Types 2-15
- Separator
 - Definition of Logical Operator, Sequential Operator, Separator 2-9
- Sequential Operator
 - Definition of Logical Operator, Sequential Operator, Separator 2-9
- Serial File
 - Serial File Parameters on CHANNEL Statement 7-1
- Side-Effect
 - Side-Effect during Evaluation of Function Designator 2-17
- Sign
 - Suppression of Sign, Zero, Comma 3-5

- Simple Variable
 - Definition of Expression, Simple Variable 2-15
 - Procedure and Simple Variable Monitoring 9-6
- SKIPB
 - Hardware Function Procedures SKIPF, SKIPB 3-50
- SKIPF
 - Hardware Function Procedures SKIPF, SKIPB 3-50
- Snap
 - Additional Delimiters Trace, Snap, Snapoff 2-10
 - Debugging Directives Trace, Snap, Snapoff 9-1
 - Debugging Directive Snap 9-3
- Snapoff
 - Additional Delimiters Trace, Snap, Snapoff 2-10
 - Debugging Directives Trace, Snap, Snapoff 9-1
 - Debugging Directive Snapoff 9-3
- Source
 - Source Listing Produced by Compilation 5-1
- Source Deck
 - Source Deck 4-4
- Source Input
 - Source Input for ALGOL Program 4-1
 - Source Input Restrictions 4-3
- Source Program
 - Definition of Source Program 4-1
 - Definition of Source Procedure 4-2
 - Use of Code in Source Procedure 4-3
- SPEC
 - Macro SPEC B-5
- Special Characters
 - Meaning of Special Characters in ALGEDIT 17-3
- Specificator
 - Definition of Bracket, Declarator, Specificator 2-10
 - Specification in a Procedure 2-49
- Stack
 - Object Time Stack 14-2
 - Example of Object Time Stack 14-4
 - Format of Entities in Object Time Stack 14-5
- Standard
 - Standard Procedures and Reserved Identifier 2-18
- Standard Format
 - Discussion of Standard Format 3-13
- Standard Procedures
 - Standard Procedures for Vector and Matrix Manipulation D-1
- Statement
 - Definition of Statement, Block Head, Block 2-27
- Step
 - Step Until Element in a For List 2-37
- STOREITEM
 - Procedures FETCHITEM, STOREITEM 3-48
- STORELIST
 - Procedures FETCHLIST, STORELIST 3-48
- String
 - Definition of String 2-14
 - Hardware Representation of String Quotes 2-14
 - Macro STRING B-11

- STRING ELEMENT
 - Procedures CHLENGTH, STRING ELEMENT 3-45
- String Format
 - Definition of String Format, Alpha Format, Boolean Format 3-7
 - Semantics of String Format, Alpha Format 3-9
- Structure
 - Structure of ALGOL 2-7
- Subscript
 - Definition of Subscript List 2-15
 - Description of Subscript and Subscripted Variables 2-16
 - Array Subscript Bounds 2-44
 - Subscript Optimization 12-1
 - Description of Subscript and Subscripted Variables 2-16
 - Array and Subscripted Variable Monitoring 9-7
 - Evaluation of Subscript Expression 2-26
- Suppression
 - Suppression of Sign, Zero, Comma 3-5
- Switch
 - Definition of Designational Expression, Label, Switch 2-26
- Switch Declaration
 - Switch Declaration 2-46
- Switch List
 - Switch List 2-46
- Symbols
 - Character Representation of ALGOL Symbols C-1
- Syntax
 - BNF (Backus Normal Form) Syntax Description 2-7
 - ALGEDIT Command Syntax 17-2
- Syntax Checking
 - Interactive Syntax Checking with ALGEDIT 17-23
- SYSPARAM
 - Procedure SYSPARAM 3-41

- T
 - Parameters C, P, T on LGO Statement 8-2
- TABULATION
 - Procedures TABULATION, NO DATA 3-21
- TEACH
 - AIDA TEACH Command 16-5
- Term
 - Definition of Arithmetic Expression, Primary, Factor, Term 2-19
- Text
 - ALGEDIT Text Replacement Command 17-15
- Text Mode
 - ALGEDIT Text Mode 17-4
- Then
 - Meaning of If, Then, Else 2-33
- THRESHOLD
 - Procedures CLOCK, THRESHOLD, INRANGE 3-52
- Title Format
 - Semantics of Title Format, Alignment Mark 3-11
- Trace
 - Additional Delimiters Trace, Snap, Snapoff 2-10
 - Debugging Directives Trace, Snap, Snapoff 9-1
 - Debugging Directive Trace 9-2

- Transfer Function
 - Transfer Function such as Entier 2-19
- Transmission
 - Transmission Error Processing 3-54
- True
 - Logical Values True and False 2-9
- Truncation
 - Truncation and Rounding Real Number on Input/Output 3-5
- Truth Table
 - Truth Table for Boolean Operator 2-25
- Type
 - Integer and Real Type 2-13
 - Type of Operand in Arithmetic Expression 2-22
 - Variables of Type Boolean 2-24
 - Definition of Type of Variable in Assignment Statement 2-31
 - Data Type Real, Integer, Boolean 2-43
 - Array Type 2-45
 - Macro TYPE B-4
- Types
 - Quantities, Kinds, Scopes, Values, Types 2-15
- Type Declaration
 - Definition of Type Declaration 2-42
- U
 - Parameters U, C, S on ALGOL Control Statement 6-3
- Unit Number
 - Unit Number and Channel Number 3-16
- UNLOAD
 - Hardware Function Procedures UNLOAD, BACKSPACE 3-51
- Until
 - Step Until Element in a For List 2-37
- Upper Bound
 - Definition of Lower Bound, Upper Bound 2-44
- Upper Case
 - Upper Case and Lower Case Letter 2-9
- VALUE
 - Macro VALUE B-6
- Values
 - Quantities, Kinds, Scopes, Values, Types 2-15
- Variable
 - Definition of Variable 2-16
 - Definition of Type of Variable in Assignment Statement 2-31
 - Controlled Variable in a For Clause 2-37
 - Displaying and Modifying Variable in AIDA 16-9
- VBOOL
 - Text of Procedures BNOT, VBOOL D-7
 - Procedures VDOT, BNOT, VBOOL, VSBOOL D-3

- VDIAD
 - Procedures VDIAD, VSDIAD, VPROSUM D-3
 - Text of Procedures VDIAD, VSDIAD D-6
- VDOT
 - Text of Procedures VPROSUM, VDOT D-7
 - Procedures VDOT, BNOT, VBOOL, VSBOOL D-3
- Vector
 - Optimization of Vector Functions 12-2
 - Standard Procedures for Vector and Matrix Manipulation D-1
 - Example of Vector Procedure Use D-9
- Vertical
 - Horizontal and Vertical Control 3-17
- VINIT
 - Procedures VINIT, VXMIT, VMONOD D-2
 - Text of Procedures VINIT, VXMIT D-4
- Virtual
 - Additional Delimiters Overlay, Virtual, Checkon, Checkoff 2-10
 - Virtual Array 2-45
 - Virtual Array 11-1
 - Example of Virtual Array 11-1
- VMONOD
 - Procedures VINIT, VXMIT, VMONOD D-2
 - Text of Procedure VMONOD D-5
- VPROSUM
 - Procedures VDIAD, VSDIAD, VPROSUM D-3
 - Text of Procedures VPROSUM, VDOT D-7
- VREL
 - Procedures VREL, VSREL D-4
 - Text of Procedures VSBOOL, VREL D-8
- VSBOOL
 - Text of Procedures VSBOOL, VREL D-8
 - Procedures VDOT, BNOT, VBOOL, VSBOOL D-3
- VSDIAD
 - Procedures VDIAD, VSDIAD, VPROSUM D-3
 - Text of Procedures VDIAD, VSDIAD D-6
- VSKIP
 - Procedures H SKIP, VSKIP 3-39
- VSREL
 - Procedures VREL, VSREL D-4
- VXMIT
 - Procedures VINIT, VXMIT, VMONOD D-2
 - Text of Procedures VINIT, VXMIT D-4
- V END
 - Procedures H LIM, V LIM, H END, V END 3-21
- V LIM
 - Procedures H LIM, V LIM, H END, V END 3-21
- W
 - Parameters C, B, I, W on CHANNEL Statement 7-1
- While
 - While Element in a For List 2-37
- Word Addressable
 - Indexed File and Word Addressable Parameters on CHANNEL Card 7-1
- WRITE ECS
 - Extended Core Storage, Large Core Memory Procedures READ ECS, WRITE ECS 3-56

X

Parameters P, E, B, D, F, K, X on ALGOL Control-Statement 6-4

XEQ

Macro XEQ B-12

XFORM

Macro XFORM B-10

Zero

Suppression of Sign, Zero, Comma 3-5

<

< and > as Metalinguistic Brackets 2-8

COMMENT SHEET



TITLE: ALGOL Version 4 Reference Manual

PUBLICATION NO. 60496600

REVISION B

This form is not intended to be used as an order blank. Control Data Corporation solicits your comments about this manual with a view to improving its usefulness in later editions.

Applications for which you use this manual.

Do you find it adequate for your purpose?

What improvements to this manual do you recommend to better serve your purpose?

Note specific errors discovered (please include page number reference).

CUT ON THIS LINE

General comments:

FROM NAME: _____ POSITION: _____

COMPANY:

NAME: _____

ADDRESS: _____

NO POSTAGE STAMP NECESSARY IF MAILED IN U.S.A.

FOLD ON DOTTED LINES AND STAPLE

STAPLE

STAPLE

FOLD

FOLD

FIRST CLASS
PERMIT NO. 8241

MINNEAPOLIS, MINN.

BUSINESS REPLY MAIL
NO POSTAGE STAMP NECESSARY IF MAILED IN U.S.A.

POSTAGE WILL BE PAID BY

CONTROL DATA CORPORATION

Publications and Graphics Division

215 Moffett Park Drive

Sunnyvale, California 94086

CUT ON THIS LINE

FOLD

FOLD

STAPLE

STAPLE